KURS PYTHON OD PODSTAW

CZĘŚĆ 1

● Programowanie w Pythonie: środowisko, podstawowe instrukcje i skrypty

Python to język programowania najbardziej ceniony w ostatnich latach. Sprawdza się w bardzo wielu zastosowaniach i ma prostą składnię. W tym kursie poznamy go od podstaw i stworzymy własne projekty

Ten kurs zawiera najważniejsze informacje o języku programowania Python – pomoże przejść od podstaw do tworzenia własnych projektów.

Pierwszy odcinek to Python w pigułce – poznamy w nim narzędzia potrzebne do pisania programów, z których będziemy korzystać w kolejnych odcinkach. Nauczymy się też tego, co najważniejsze, o podstawowych instrukcjach w trybie konsolowym, takich jak pętle. Dowiemy się, czym są moduły i jak wykorzystać dodatkowe zestawy poleceń, by tworzyć bardziej rozbudowane projekty.

W drugim odcinku motywem przewodnim będzie wykorzystanie okna graficznego.

Odcinek trzeci zostanie poświęcony modułowi Pygame – poznamy bibliotekę do tworzenia gier, pisząc krok po kroku prostą grę graficzną.

Odcinek czwarty to Python w edukacji informatycznej – przeczytamy, jakie moduły Pythona i do czego są wykorzystywane w szkołach, w programie nauczania.

W piątym odcinku dowiemy się, jak wykorzystywać bazy danych z poziomu Pythona, używając modułu sqlite3.

kj

TEMATY ODCINKÓW KURSU

1. Python: co musisz wiedzieć na początek
2. Programy z oknem graficznym
3. Pygame – moduł do tworzenia gier
4. Python jako język edukacyjny
5. Wykorzystanie baz danych w programach napisanych w języku Python

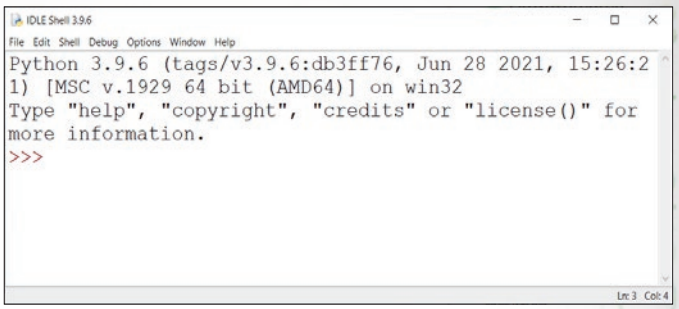
Fot. Photo5963/iStockphoto.com

IDLE – PODSTAWOWE ŚRODOWISKO PYTHONA

Narzędzia języka Python są instalowane domyślnie wraz z oprogramowaniem IDLE. To proste IDE (czyli zintegrowane środowisko

programistyczne) przeznaczone jest właśnie do programowania w Pythonie. Ciekawostką jest to, że program ten sam został stworzony właśnie w Pythonie.

To, co zobaczymy po uruchomieniu tego oprogramowania, przypomina konsolę, czyli tryb tekstowy. Programy, które nie mają własnego okna graficznego, a ich działanie ogranicza się do wyświetlania informacji tekstowych i pobierania ich z poziomu okna konsoli,



nazywamy konsolowymi. A tę konsolę narzędzia IDLE nazywamy **Shell**. Proste programy możemy pisać nawet bezpośrednio w konsoli. To dlatego, że Python jest językiem **interpretowanym**. To znaczy, że każda linijka wprowadzonego przez nas kodu jest wykonywana od razu po uruchomieniu programu – a w tym przypadku po wprowadzeniu jej do konsoli, niezależnie od tego, jaki mamy zamysł na pozostałą treść programu.

PYTHON JAKO KALKULATOR

Wprowadzanie instrukcji w oknie konsoli jest dobre choćby w wypadku obliczeń matematycznych. Jeśli chcemy potraktować Pythona jako kalkulator, możemy to zrobić tak.

1 Wpisujemy w oknie konsoli treść działania matematycznego, na przykład **6 + 4**.

```
>>> 6 + 4
10
>>>
```

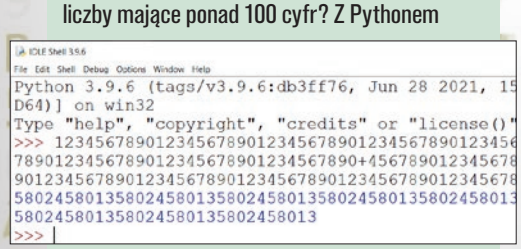
2 Po treści działania wciskamy na klawiaturze enter, wywołując w ten sposób wykonanie wprowadzonej przez nas linii kodu.

Uwaga! Działania nie kończymy znakiem równości, ponieważ taki zapis skutkuje pojawieniem się w oknie konsoli informacji o błędzie.

```
>>> 6 + 4 =
SyntaxError: cannot assign to operator
>>>
```

Obliczenia na ogromnych liczbach

Zaletą Pythona jest to, że doskonale radzi sobie z działaniami na ogromnych liczbach, które znacznie przekraczają zakres zmiennych w wielu innych językach. Chcemy dodać do siebie liczby mające ponad 100 cyfr? Z Pythonem

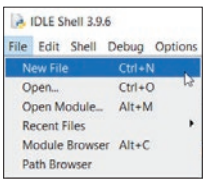


to żaden problem, poradzi sobie z naprawdę bardzo dużymi liczbami.

OSOBNE PLIKI Z KODEM

Tryb wprowadzania instrukcji bezpośrednio do okna konsoli jest wygodny tylko w wypadku tworzenia niewielkich programów. Kiedy skrypty zawierają więcej linii, a my chcemy mieć realny wpływ na edycję wcześniej napisanych treści, choćby w wypadku popełnienia błędu, można przygotować cały plik ze skryptem i uruchomić dopiero po napisaniu całości.

1 W oknie konsoli rozwijamy menu, klikając na **File**, i wybieramy pozycję **New File**.

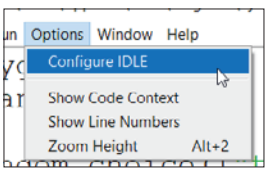


2 Otwarte zostanie okno, które przypomina okno programu Notatnik. Ma ono jednak znacznie większe

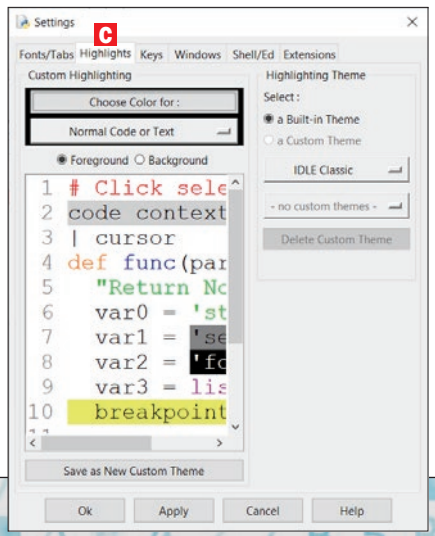
możliwości. Pierwszą z nich jest możliwość kolorowania składni **A**. Jest to podstawowa i najważniejsza funkcja edytorów kodu. Już podczas pisania, na podstawie tego,

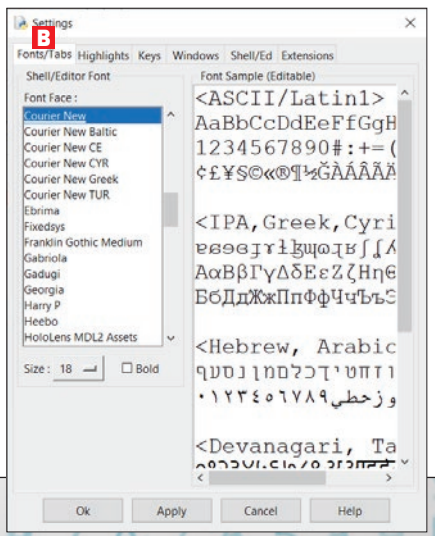
jak kolorowane są poszczególne fragmenty, możemy weryfikować, czy nie popełniliśmy błędu w skrypcie.

3 To, jak wygląda przedstawiony w oknie tekst, możemy zmienić. W tym celu w menu górnym klikamy na **Options** i wybieramy pozycję **Configure IDLE**.



4 Otwiera się okno podzielone na zakładki. W pierwszej z nich – **Font/Tabs** **B** – możemy zmienić rodzaj czcionki i jej wielkość. A kolory, na jakie podświetlana jest składnia, możemy zmienić z poziomu zakładki **Highlights** **C**.

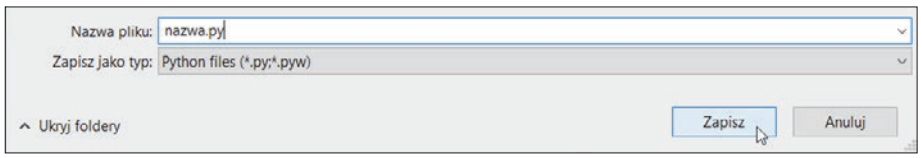




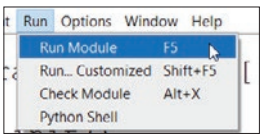
Wyrwij, użyj dziurkacza i zbieraj do segregatora

URUCHAMIANIE

Jeśli napiszemy już pierwsze skrypty, musimy wiedzieć, jak je uruchomić.



1 Z menu górnego rozwijamy opcję **Run**, a następnie wybieramy **Run module** lub naciskamy klawisz F5.



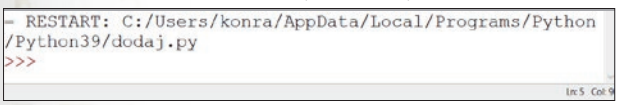
2 Pamiętajmy, że skrypt przed uruchomieniem musi być zapisany – jeśli zapis nie został wykonany, program wyświetli stosowne okno, abyśmy mogli tego dokonać. Skrypty języka Python zapisujemy z rozszerzeniem **.py**.

PODSTAWOWE INSTRUKCJE

OPERACJE WEJŚCIA I WYJŚCIA

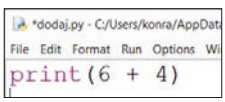
Operacje wejścia i wyjścia to podstawowy element komunikowania się programu i użytkownika. Jeśli w skrypcie zapisanym w formie pliku tekstowego pojawi się działanie – na przykład takie, jakie we wcześniej opisanym przykładzie wpisaliśmy w oknie konsoli, czyli: **6 + 4** – to po uruchomieniu skryptu wynik tego

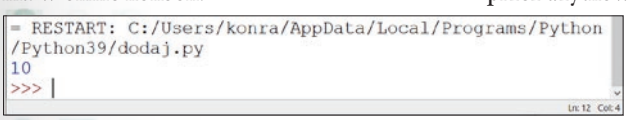
działania nie pokaże się od razu w oknie. Okno Shell pozostanie bez wypisanej informacji.



WYJŚCIE DANYCH

Do wyświetlenia informacji w konsoli służy specjalne polecenie **print()** – to, co wpiszemy w nawiasie, to treść do wyświetlenia. Zatem zapis **print(6 + 4)** powinien po uruchomieniu programu wyświetlić wynik dodawania w oknie konsoli.





WEJŚCIE DANYCH

Programy pisane przez nas w trybie konsolowym mogą też pobierać informacje od użytkownika, aby następnie je przetworzyć. Przyjmując, że chcielibyśmy stworzyć program, którego zadaniem będzie przeliczenie temperatury określonej w stopniach Celsjusza na temperaturę wyrażoną w stopniach Fahrenheita – nasz program powinien dostać od użytkownika informację o tym, jaką temperaturę ma przeliczyć.

Co więcej, program powinien tę wartość temperatury odpowiednio przechować.

Do przechowywania informacji służą **zmienne**. Czym one są? Zmienne w programowaniu najłatwiej porównać do pojemnika, w którym można przechowywać dane. Operując na danych w skrypcie, wcale nie musimy znać dokładnych wartości, ponieważ odnosimy się do nich poprzez ich nazwę. Tak więc zmienna to pewna wartość, którą przechowujemy pod określoną nazwą.

W przypadku programu do przeliczania temperatur wartość temperatury w stopniach Celsjusza moglibyśmy przechować w zmiennej o nazwie **celsjusze**. By ją utworzyć, wystarczy, że użyjemy w kodzie źródłowym jej nazwy. Z kolei by nadać zmiennej wartość zmiennej o określonej w ten sposób nazwie, należy zapisać ją po znaku równości. Zatem zapis **celsjusze = 32** tworzy zmienną o nazwie **celsjusze**, której wartość to **32**. Nie jest to jednak wartość podana przez użytkownika programu.

By zmienna miała wartość nadaną przez użytkownika, należałoby zamiast konkretnej liczby użyć polecenia **input()**. Polecenie to pozwala nam nie tylko odczytać wartość od użytkownika, ale też wcześniej wyświetlić użytkownikowi komunikat, który może mu wyjaśnić, jakie dane trzeba wprowadzić do programu.

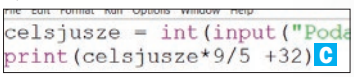
Jeśli użytkownik ma podać wartość temperatury do przeliczenia, możemy zapisać w skrypcie linię: **celsjusze = input("Podaj temperaturę w stopniach Celsjusza")** A. I faktycznie, taki zapis będzie działał poprawnie, użytkownik wprowadzi wartość dla zmiennej – jednak program nie rozpozna tej wartości jako liczby, a co za tym idzie, nie będzie mógł wykorzystać jej do obliczeń.

By było to możliwe, polecenie **input** powinno znaleźć się w nawiasie po poleceniu **int()**. Zatem cała i pełna już linia kodu powinna mieć postać **celsjusze = int(input("Podaj temperaturę w stopniach Celsjusza"))** B.

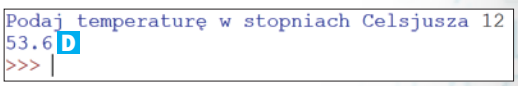
Z tak podaną wartością zmiennej możemy już przeprowadzać obliczenia.

Wynik chcemy pokazać na wyjściu danych, a więc trzeba użyć polecenia **print**.

By przeliczyć podaną przez użytkownika temperaturę w stopniach Celsjusza na stopnie Fahrenheita, powinniśmy skorzystać ze wzoru **F = (C * 9/5) +32** – co w naszym kodzie zapiszemy jako: **print(celsjusze*9/5 +32)** C.



Gdy uruchomimy już tak zapisany skrypt, wyświetli on prośbę o podanie temperatury. Wpisujemy liczbę i zatwierdzamy, wciskając klawisz enter. Program dalej wyświetla linię D z obliczoną według wzoru wartością.



PĘTLE

Ważnym elementem Pythona i innych języków programowania są pętle. Służą one do wielokrotnego wykonywania zawartych w nich instrukcji. Dwie najważniejsze pętle w językach programowania to **for** i **while**. Możemy powiedzieć, że pierwsza z nich pozwala nam wykonać czynność określoną liczbę razy, druga zaś – wykonywać coś aż do osiągnięcia określonego celu.

PĘTLA FOR

1 Załóżmy, że chcemy wypisać pięć kolejnych liczb – od 1 do 5. Wypisanie powinno wykonać się pięć razy. Użyjemy do tego pętli **for**. By wykonała ona pięć powtórzeń, zapisujemy **for** i **in range(5):**

```
for i in range(5):
```

2 Aby zapisać instrukcję, która ma się zapętlić, należy przejść do kolejnej linii kodu. IDLE automatycznie ustawi kursor z wcięciem. Dzieje się tak, ponieważ treść pętli musi być w jakiś sposób wydzielona. W wielu językach programowania stosuje się do wydzielania treści nawiasy klamrowe – tak jest w językach C#, C++, Java czy JavaScript. Z Pythonem jest nieco inaczej. Zapętlone będą wszystkie linie kodu napisane pod pętlą, jeśli są poprzedzone wcięciem – aż do pierwszej napotkanej w kodzie linii zapisanej bez wcięcia. W przypadku wypisywania zapętlona powinna być instrukcja **print**.

```
for i in range(5):
    print()
```

3 Zapętlona instrukcja ma wypisywać liczby od 1 do 5 – musimy więc zastanowić się, skąd je wziąć. Z pomocą przychodzi tu sama pętla. W jej treści pojawia się litera **i**, która tak naprawdę jest zmienną przyjmującą w obrębie pętli wartości kolejnych liczb, rozpoczynając od 0. Z każdym przejściem pętli ma ona wartość o 1 większą. Są to wartości 0, 1, 2, 3 i 4. Jeśli chcemy wypisać kolejno liczby 1, 2, 3, 4 i 5 – możemy wypisywać wartość działania **i + 1**.

```
for i in range(5):
    print(i+1)
```

4 Efekt działania takiego skryptu będzie zgodny z założeniami zadania.

```
= RESTART: C:/Users
1
2
3
4
5
>>>
```

PĘTLA WHILE

Drugi z rodzajów pętli omówimy na przykładzie przedstawionego skryptu.
Program opisany tym skryptem będzie prosił użytkownika o podanie liczby, dopóki nie trafi on na tę, która okaże się poszukiwaną wartością. Wtedy program pogratuluje i wypisze, ile prób wykonał użytkownik, zanim trafił na poszukiwaną liczbę.

```
A podana = 0
B szukana = 77
C licznik = 0
D while szukana != podana:
    E podana = int(input("Podaj liczbę: "))
    F licznik = licznik + 1
G print("Brawo! podano prawidłową liczbę. Wykonano prób: "+str(licznik))
```

A Deklarujemy zmienną, w której będziemy przechowywać liczby podawane przez użytkownika.

B Deklarujemy zmienną przechowującą poszukiwaną liczbę, której podanie zakończy działanie programu.

C Deklarujemy zmienną do przechowywania liczby prób podejmowanych przez użytkownika.

D Tworzymy pętlę **while**, która zawiera warunek wykonywania pętli – warunkiem tym jest to, że podana przez użytkownika zmienna ma wartość inną niż szukana (zapis **!=** jest równoznaczny z tym stwierdzeniem).

E Linia zapisana z wcięciem wykonuje się w pętli – gdy warunek pętli jest prawdą. W linii tej użytkownik nadaje zmiennej podaną nową wartość.

F Ta linia również wykonuje się w pętli, ponieważ jest zapisana z wcięciem. W linii tej zwiększamy o jeden wartość licznika prób.

G Ta linia wykonuje się tylko raz – nie jest zapisana z wcięciem, więc wykona się dopiero wtedy, kiedy pętla skończy powtarzać zapętlone instrukcje. W tej linii wypisujemy informację dla użytkownika, że udało mu się trafić i podać poszukiwaną liczbę oraz ile prób wykonał.

A oto przykładowe użycie programu:

```
= RESTART: C:/Users/konra/AppData/Local/Programs/
Podaj liczbę: 34
Podaj liczbę: 23
Podaj liczbę: 65
Podaj liczbę: 77
Brawo! podano prawidłową liczbę. Wykonano prób: 4
>>>
```

INSTRUKCJA WARUNKOWA

Instrukcja warunkowa to kolejny podstawowy element języków programowania. Pozwala ona na wykonywanie pewnych poleceń tylko wtedy, kiedy spełniony jest określony w niej warunek. To mechanizm podobny do pętli **while**, jednak nie występują w niej powtórzenia. Podobnie jak w przypadku pętli, instrukcje, które mają wykonać się tylko wtedy, gdy podany warunek jest spełniony, zapisujemy z wcięciem względem linii z samym warunkiem. Warunek zaś zapisujemy po słowie **if**. Oto przykładowa instrukcja warunkowa **A** – linia **print("A jest mniejsze niż B")** wykona się tylko wtedy, gdy zmienna **a** ma wartość mniejszą niż zmienna **b**.

```
a = 10
b = 11
if a<b: A
    print("A jest mniejsze niż B")
```

MODUŁY

Wiele poleceń języka Python jest umieszczanych w modułach. Część takich zbiorów to moduły podstawowe, instalowane wraz z całym pakietem języka Python. Takie moduły możemy dodać (importować), po prostu używając w kodzie słowa **import** i podając nazwę modułu. Są jednak także moduły, które musimy najpierw zainstalować, aby móc później importować je do projektu. Wśród modułów podstawowych można wymienić: **tkinter** – moduł dający dostęp do okna graficznego (bo w Pythonie możemy tworzyć nie tylko programy konsolowe, ale też takie z oknem graficznym), **random** – moduł zawierający polecenia pozwalające generować liczby

Wyrwij, użyj dziurkacza i zbieraj do segregatora

z określonych przedziałów, **math** – zawierający wiele funkcji matematycznych przydatnych do wykonywania obliczeń.
Korzystając z importowanych w ten sposób modułów i wywołując zasoby w nich zawarte, najpierw podaje się nazwę modułu, a następnie po kropce nazwę wykorzystywanego zasobu.
Istnieje wiele modułów, których wykorzystanie determinuje to, jak wyglądają same skrypty, ale też dają one ogrom możliwości, dostarczając rozwiązania pozwalające na budowę zaawansowanych aplikacji.
Stosując przykładowe, popularne moduły, można tworzyć na przykład gry.

PYGAME

Jest to moduł do tworzenia gier. Daje nam dostęp do okna graficznego i zapewnia bogaty zestaw poleceń pozwalających na umieszczanie w oknach rozmaitych obiektów oraz kontrolowanie ich. Warto wiedzieć, że na stronie **pygame.org** jest opisanych i zaprezentowanych wiele projektów stworzonych z wykorzystaniem tego modułu. Wśród nich znajdziemy gry takie jak **BoxeD** A czy **Trouble In CloudLand** B. Pygame dobrze sprawdza się w tworzeniu gier w stylu retro.

PANDA3D

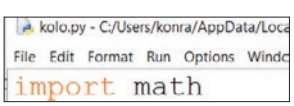
Panda3D to z kolei moduł, który można wykorzystać do gier z trójwymiarową grafiką. Dostarcza nie tylko zestaw poleceń umożliwiających łatwiejsze obsługiwanie obiektów trójwymiarowych, ale nawet trójwymiarowy model pandy, której ruchem możemy zarządzać poprzez skrypty.

ĆWICZENIA PRAKTYCZNE
SKRYPTY KROK PO KROKU

POLE KOŁA I OBWÓD OKRĘGU

Po poznaniu podstaw składni języka i obsługi narzędzia można przejść do realizacji zadań. Pierwszym z zadań będzie napisanie **programu do obliczania pola koła**. By je obliczyć, potrzebujemy wiedzieć, jaki jest promień koła – ponieważ występuje on we wzorze na pole koła. Promień należy podnieść do kwadratu i pomnożyć przez wartość liczby pi.

1 I właśnie na tej liczbie skupimy się, przystępując do pracy nad programem. Python daje nam możliwość skorzystania w obliczeniach ze stałej przechowującej wartość liczby pi. Jest ona jednak częścią modułu o nazwie **math**. Pisanie skryptu rozpoczniemy więc od importowania tego modułu, czyli od **import math**.



2 By przeprowadzić mnożenie, wykorzystując wartość liczby pi z importowanego modułu, potrzebujemy pozostałych wartości, tu będzie to promień koła. Jego wartość program powinien otrzymać od użytkownika i zapisać w zmiennej. Jeśli zmienna ta ma mieć wartość **r**, powinniśmy użyć linii kodu: **r = int(input("Podaj promień: "))**.

```
import math
r = int(input("Podaj promień: "))
```

3 Dalej można już wykonać obliczenia. Wartość liczby pi z modułu **math** otrzymamy, zapisując ją jako **math.pi**. Następnie musimy pomnożyć ją przez kwadrat promienia, czyli dwukrotnie pomnożyć przez promień. By wynik umieścić w zmiennej o nazwie **p**, zapisujemy linię kodu: **p = math.pi*r*r**.

```
p = math.pi*r*r
```

4 Wynik działania zapisany w zmiennej o nazwie **p** powinniśmy jeszcze wypisać, korzystając z polecenia **print**. Oprócz samej wartości liczbowej użytkownik programu powinien zobaczyć też informację o tym, co reprezentuje wypisana wartość. Całość linii kodu może mieć postać: **print("Pole koła: "+str(p))**.

```
import math
r = int(input("Podaj promień: "))
p = math.pi*r*r
print("Pole koła: "+str(p))
```

Dla podanego promienia **możemy obliczyć też długość okręgu, czyli obwód**. Jego wartość obliczamy, mnożąc promień przez 2 i przez pi.

1 Deklarujemy zmienną o nazwie **ob** i nadajemy jej wartość będącą wynikiem mnożenia, pisząc: **ob = 2*math.pi*r**.

```
print("Pole koła: "+str(p))
ob = 2*math.pi*r
```

2 Podobnie jak w przypadku pola koła wynik obliczeń należy wypisać wraz z informacją, co reprezentuje wypisywana liczba.

```
print("Pole koła: "+str(p))
ob = 2*math.pi*r
print("Obwód okręgu: "+str(ob))
```

Można to zrobić, umieszczając w kodzie źródłowym linię: **print("Obwód okręgu: "+str(ob))**.

Gotowy program można już przetestować. Dla raz wprowadzonego promienia powinien on wyświetlić zarówno pole **A**, jak i obwód **B** obliczone ze wspomnianych wzorów.

```
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:\Users\konra\AppData\Local\Programs\Python\Python39\kolo.py ====
Podaj promień: 4
Pole koła: 50.26548245743669 A
Obwód okręgu: 25.132741228718345 B
>>>
```

Import modułów

W przedstawionych zadaniach jednym z ważnych elementów był import modułów. Do tej pory omawialiśmy tylko jeden sposób importu modułów. Trzeba użyć słowa **import** i podać po nim nazwę modułu. A żeby skorzystać z zawartości modułu, podajemy jego nazwę i po kropce określamy, jaka zawartość modułu nas interesuje.
Można skorzystać z innego sposobu: zawartość modułu przenosimy do przestrzeni nazw projektu i nie musimy podawać nazwy modułu, korzystając z jego zawartości. Importując w ten sposób moduł **math**, należy zapisać **from math import ***. Wtedy w pierwszym zadaniu zamiast pisać **math.pi**, można byłoby dostać się do stałej, pisząc tylko **pi**.

MINUTNIK

Minutnik to program do odliczania zadanego czasu i informowania użytkownika, że określony czas już upłynął. W tym przykładzie stworzymy program, którego zadaniem będzie odliczenie 60 sekund.
Tym razem także niezbędne jest skorzystanie z modułu. Będzie to moduł o nazwie **time**.

1 I tym razem należy rozpocząć pisanie skryptu realizującego omawiane zadanie od **import time**.

```
File Edit Format Run Opti
import time
```

2 W kolejnym kroku możemy utworzyć zmienną do przechowania wartości czasu, jaki należy odliczyć. Jeśli ma to być jedna minuta – powinna być ona wyrażona w sekundach, zatem zmienna powinna otrzymać wartość **60**.

```
import time
do_odliczenia = 60
```

3 Podany czas powinniśmy co sekundę zmniejszać i wypisywać użytkownikowi na wyjściu danych. Jeśli operacja ta ma wykonać się wiele razy, to znaczy, że powinna zostać zapisana w pętli. Może to być pętla **for**, która wykona się tyle razy, jaka jest wartość zmiennej przechowującej czas do odliczenia.

```
import time
do_odliczenia = 60
for i in range(do_odliczenia):
```

4 Wewnątrz pętli mamy dostęp do zmiennej **i**, której wartość rośnie od zera z każdym przejściem pętli. Jeżeli wartość tę będziemy odejmować od czasu do odliczenia i w każdym przejściu pętli wypisywać ten wynik poleceniem **print** – uzyskamy efekt odliczania czasu.

```
*minutnik.py - C:/Users/konra/AppData/Local/Programs/P...
File Edit Format Run Options Window Help
import time
do_odliczenia = 60
for i in range(do_odliczenia):
    print(do_odliczenia - i)
```

```
for i in range(do_odliczenia):
    print(do_odliczenia - i)
    time.sleep(1)
```

5 Liczby będą wypisywane – jednak w tym momencie pętla wykonuje się tak szybko, jak tylko pozwalają jej na to zasoby sprzętowe. Nam natomiast zależało na uzyskaniu rzeczywistego odliczania czasu. By każde wypisywanie następowało po sekundzie, należy wykorzystać polecenie z importowanego na początku skryptu modułu **time**. Mowa tu o poleceniu **sleep()**, w którym podajemy, na ile sekund ma się zatrzymać działanie programu. Jeśli wewnątrz pętli umieścimy linię kodu **time.sleep(1)**, przejścia pętli będą wykonywać się co sekundę.

6 Po zakończeniu działania pętli możemy już wypisać użytkownikowi informację o tym, że czas upłynął. Jeśli polecenie ma wykonać się po zakończeniu działania pętli **for**, należy zapisać je bez wcięcia na początku linii.

```
*minutnik.py - C:/Users/konra/AppData/Local/Programs/P...
File Edit Format Run Options Window Help
import time
do_odliczenia = 60
for i in range(do_odliczenia):
    print(do_odliczenia - i)
```

LICZNIK LITER

Teraz napiszemy program, który w podanym słowie będzie zliczał wystąpienia wybranej przez nas litery.

1 Zapisujemy słowo i litery do zliczania; powinny one być pobrane od użytkownika, za co odpowiada polecenie **input** **A**.

2 Liczbę wystąpień też przechowamy w zmiennej – niech nazywa się **licznik**, a jej wartością początkową będzie **0** **B**.

3 By przeanalizować wszystkie litery w podanym słowie, skorzystamy z pętli **for**. Wykorzystamy ją inaczej niż w poprzednim przykładzie. Słowo range zastąpimy nazwą zmiennej – **slowo**. W ten sposób zapis **for i in slowo:** **C** sprawi, że **i** jako zmienna w pęt li z każdym przejściem pętli będzie przyjmować wartość kolejnych znaków w słowie.

```
*licznikLiter.py - C:/Users/konra/AppData/Local/Programs/Python/Python39/licznikLiter.py (3.9.6)*
File Edit Format Run Options Window Help
slowo = input("Podaj słowo: ") A
litera = input("Podaj literę: ")
licznik = 0 B
for i in slowo: C
  D if i == litera:
        licznik = licznik + 1 E
print("Liczba wystąpień litery "+litera+" w słowie "+slowo+" to:"+str(licznik)) F
```

4 Poprzez instrukcję warunkową **if** **D** w pęt li sprawdzimy, czy każda kolejna litera jest taka jak ta podana przez użytkownika.

5 Jeśli warunek z instrukcji warunkowej jest spełniony, zwiększamy wartość licznika o 1, pisząc **licznik = licznik + 1** **E**. Na koniec wystarczy przez polecenie **print** wypisać informację dla użytkownika; może się ona składać z fragmentów tekstu i wartości zmiennych – takie elementy informacji łączymy plusem **F**.
Oto efekt działania programu:

```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python39/licznikLiter.py
Podaj słowo: kolaboracja
Podaj literę: a
Liczba wystąpień litery a w słowie kolaboracja to:3
>>>
```

Wyrwij, użyj dziurkacza i zbieraj do segregatora

PYTHON – PROGRAMY Z OKNEM GRAFICZNYM

Wiemy już, że Python jest językiem chętnie wykorzystywanym w szkołach jako pierwszy poznawany język programowania. W poprzednim odcinku kursu poznaliśmy już podstawy tego języka. Z tego odcinka dowiemy się, jak korzystając z języka Python, tworzyć programy z oknem graficznym

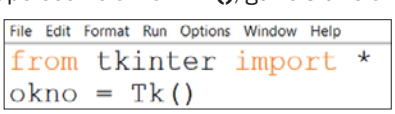
OKNO GRAFICZNE Z MODUŁEM TKINTER

Do tworzenia programów z oknem graficznym można wykorzystać różne moduły. Tym podstawowym modułem jest **tkinter**. Warto wiedzieć, że program IDLE, który jest wykorzystywanym w tym kursie IDE, czyli środowiskiem programistycznym, też został stworzony z wykorzystaniem tego modułu. W tym odcinku kursu skupimy się na dwóch modułach dostarczających nam okno graficzne – wspomnianym module **tkinter** i module **turtle**. W tym drugim stworzymy między innymi grę **Wyścig żółwi**.

TWORZYMY OKNO GRAFICZNE

1 Pierwszym krokiem, by utworzyć okno graficzne, korzystając z modułu **tkinter**, jest jego import – importujemy moduł, pisząc **from tkinter import ***.

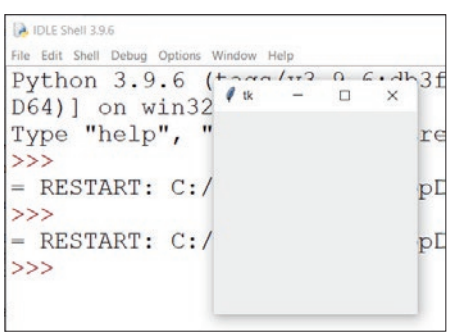
2 Drugi krok to utworzenie okna poprzez polecenie **okno = Tk()**, gdzie słowo **okno**



jest nazwą obiektu reprezentującego okno programu. Można tu użyć innej nazwy, jednak należy pamiętać, by konsekwentnie trzymać się użytego nazewnictwa w całym swoim skrypcie. Dla utworzonego okna dodajemy jeszcze wywołanie metody **mainloop**, które zawsze powinno być na końcu skryptu, a kolejne polecenia powinny być dopisywane przed nim. Można powiedzieć, że jest to swego rodzaju przejście do trybu oczekiwania na akcję użytkownika.

```
from tkinter import *
okno = Tk()


okno.mainloop()
```

3 Jeśli w tym momencie uruchomimy zapisany skrypt, oprócz okna konsoli zobaczymy niewielkie okno graficzne programu.



4 Kolejny krok powinien polegać na umieszczeniu w oknie odpowiedniej zawartości. Elementy budujące okno to tak zwane **kontrolki**. Jednym z podstawowych elementów są etykiety – kontrolki wyświetlające napisy. Podobnie jak samo okno, tak i znajdujące się w nim elementy powinny mieć swoje nazwy, abyśmy mogli odnosić się do nich z poziomu innych części kodu źródłowego programu. I tak poprzez zapisanie **etykieta = Label(okno, text='Napis')** utworzymy

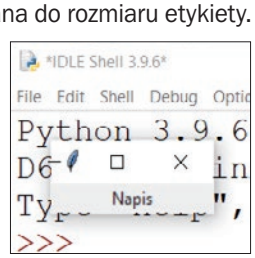
```
from tkinter import *
okno = Tk()

etykieta = Label(okno, text="Napis")
```

w oknie obiekt o nazwie **etykieta**, na którym znajdzie się tekst **Napis**. Słowo **Label** w tym poleceniu jest nazwą rodzaju kontrolki, specyficzną właśnie dla etykiet. Jeśli chcielibyśmy stworzyć przycisk, słowo to powinno być zastąpione słowem **Button**.

5 Zapisane do tej pory polecenia nie umieszczają jeszcze etykiety w oknie programu. By ta się w nim pojawiła, należy dla kontrolki wywołać metodę **pack**. Dopasuje ona wielkość etykiety do długości napisu. Program

```
etykieta = Label(okno, text="Napis")
etykieta.pack()
okno.mainloop()
```

wyświetli okno z etykietą, a jego wielkość będzie już dopasowana do rozmiaru etykiety. Zatem okno programu zapisanego jak w przykładzie zmniejszy się w stosunku do pustego okna, jakie zostało utworzone wcześniej.



Fot. Photo5963/iStockphoto.com

PRZYCISK I REAKCJA NA KLIKNIĘCIE

Zgodnie z przedstawionym schematem, w oknie programu można umieszczać też inne kontrolki – w tym przyciski.

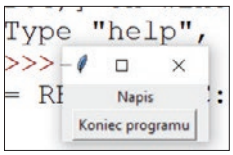
1 Jeśli chcemy, by pod etykietą pojawił się przycisk, umieszczamy w kodzie polecenie: **przycisk = Button(okno, text = "Koniec programu")**. W ten sposób zostanie utworzony przycisk z napisem **Koniec programu**.

```
etykieta = Label(okno, text="Napis")
przycisk = Button(okno, text = "Koniec programu")
etykieta.pack()
okno.mainloop()
```

Jednak, podobnie jak w przypadku etykiety, by przycisk ten był w oknie widoczny, należy dopisać dla niego wywołanie metody **pack**.

```
przycisk = Button(okno, text = "Koniec programu")
etykieta.pack()
przycisk.pack()
okno.mainloop()
```

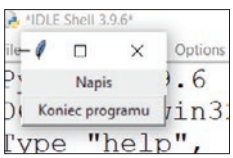
2 W oknie programu zobaczymy już niewielki przycisk.

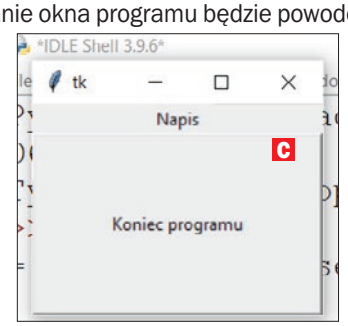


3 Metoda **pack** pozwala na określenie opcji jej działania. Robimy to w nawiasie metody – tak jak w wypadku ustawiania tekstu na przycisku i etykiecie. Nadanie atrybutowi **expand** wartości **YES** A nakazuje powiększanie kontrolki w przypadku zmiany rozmiaru okna. Natomiast wartość **BOTH** B w atrybucie **fill** oznacza, że kontrolka ma zajmować całą dostępną powierzchnię okna w obu kierunkach, czyli w pionie i poziomie. By podobny efekt wypełnienia uzyskać tylko w pionie, zamiast BOTH należy użyć wartości **y**, a tylko w poziomie – wartości **x**.

```
etykieta.pack()          A          B
przycisk.pack(expand=YES, fill=BOTH)
```

4 Po takich zmianach przycisk wypełni już wolne miejsce w okne programu, jednak nadal będzie on niewielki. Ręczne powiększanie okna programu będzie powodowało jednak, że wraz z nim powiększy się przycisk C. Efekt ten nie jest jednak dobrze widoczny, ponieważ przycisk i etykieta mają takie same kolory.





5 Podczas tworzenia elementów okna możemy określać nie tylko znajdujący się na nich tekst, ale też inne właściwości. Jedną z nich jest kolor tła kontrolki, który ustawiamy za pomocą atrybutu **background** D. Jako wartość przypisujemy do niego kolor tła, który może być zapisany w języku angielskim. Analogicznie możemy też zmienić kolor samego tekstu, używając do tego atrybutu **foreground**.

```
przycisk = Button(okno, text = "Koniec programu", background="red")
```
D

6 Sam przycisk w oknie programu to jednak zbyt mało. Dodając do okna przycisk, chcemy, by wywoływał jakieś działanie, a zatem musimy przypisać do kliknięcia na niego wykonanie jakiejś funkcji. W przypadku naszego programu stworzymy funkcję zamykającą okno graficzne. Nadamy jej nazwę **zamknij**.

```
from tkinter import *
okno = Tk()

def zamknij():
    
```

7 W jej treści umieścimy dwa polecenia: najpierw przerwanie trybu nasłuchiwania zdarzeń, czyli **okno.quit()**, a następnie zamknięcie okna i zwolnienie zajętych przez nie zasobów, czyli **okno.destroy()**.

```
def zamknij():
    okno.quit()
    okno.destroy()
```

8 Utworzona funkcja nie działa, ponieważ nie jest wywoływana. Należałoby tak zmodyfikować skrypt, aby funkcja ta była wywoływana zawsze w momencie kliknięcia na przycisk.
By tak było, należy użyć metody **bind**, która pozwala przypisać akcję do wystąpienia konkretnego zdarzenia. W jej parametrze najpierw podajemy typ zdarzenia, gdzie **"<Button>"** oznacza kliknięcie na przycisk, a następnie nazwę funkcji, jaka ma być wtedy wykonana, czyli **zamknij** E. Funkcja, która jest przypisana do zdarzenia, musi przyjmować jeden parametr. Jego nazwę umieszczamy w nawiasie przy definicji funkcji. Z tego parametru w obrębie funkcji możemy potem odczytywać różne właściwości dotyczące samego zdarzenia.

```
przycisk.pack(expand=YES, fill=BOTH)
przycisk.bind("<Button>", zamknij)E
okno.mainloop()
```

9 Testując opracowane rozwiązanie, zauważymy, że przycisk reaguje zamknięciem okna programu, gdy klikniemy na niego, zarówno używając prawego, jak i lewego przycisku myszy.
Zazwyczaj klikamy, używając lewego przycisku myszy, a prawym posługujemy się do otwarcia różnego rodzaju menu kontekstowych. Używając kontrolek z modułu **tkinter**, możemy tak sformułować zdarzenie,

```
def zamknij(parametr):
    okno.quit()
    okno.destroy()
```

by przycisk reagował tylko na kliknięcie na wybrany przycisk myszy. By był to przycisk lewy, w poleceniu przypisującym funkcję do zdarzenia zamieniamy **"<Button>"** na **"<Button-1>"**.

10 Analogicznie możemy przypisać wykonywanie innych funkcji do kliknięć innych przycisków myszy.
I tak – zdarzenie opisane jako **"<Button-2>"** będzie wywoływało określoną funkcję w momencie naciśnięcia kółka myszy.
A zdarzenie określone jako **"<Button-3>"** będzie wywoływało przypisaną funkcję w momencie wciśnięcia prawego przycisku myszy.

Funkcje

Aby zaprogramować działanie przycisku, powinniśmy wiedzieć, czym są funkcje i jak je deklarować w języku Python.
Możemy przyjąć, że funkcje to są zdefiniowane fragmenty kodu, które mają nazwy, a wykonają się wtedy, gdy w innej części skryptu nazwa funkcji zostanie użyta, czyli wywołana.
- Do definicji funkcji używa się słowa kluczowego **def**.
- Po nim podaje się nazwę tworzonej funkcji i nawias.
- W nawiasie mogą znajdować się parametry funkcji, czyli zmienne, którym wartości możemy nadać w momencie wywoływania funkcji.
- Po nawiasie umieszczamy dwukropek, który ma duże znaczenie. Jak już wiemy z poprzedniego odcinka kursu, w Pythonie treść wydzielonych fragmentów kodu po dwukropku zapisujemy z wcięciem, jak w przypadku pętli czy instrukcji warunkowej.

Wyrwij, użyj dziurkacza i zbieraj do segregatora

OKNO GRAFICZNE Z MODUŁEM TURTLE

Moduł **tkinter** stał się podstawą, na której powstały też inne moduły dające nam dostęp do okna graficznego. Jednym z nich jest moduł **turtle**. Jest on chętnie wykorzystywany w nauce programowania na poziomie szkół podstawowych i mocno odnosi się do języka LOGO, który został zaprojektowany pod koniec lat sześćdziesiątych XX wieku przez pracującego na MIT Seymoura Paperta. Ten zaś inspirował się pracami szwajcarskiego psychologa i pedagoga Jeana Piageta. Język LOGO miał mieć głównie charakter edukacyjny. Podobnie jak LOGO moduł **turtle** daje nam do dyspozycji okno graficzne, w którym możemy umieścić grafikę żółwia i dowolnie sterować nią za pomocą zestawu poleceń. Żółw ten może zostawiać za sobą ślady i w ten sposób tworzyć obrazy.

Polecenie	Opis
fd() | forward()	Przesuwa żółwia o podaną w argumencie liczbę kroków do przodu.
backward() | bk() | back()	Przesuwa żółwia o podaną w argumencie liczbę kroków w tył.
lt() | left()	Obraca żółwia w lewą stronę o podany w argumencie kąt (w stopniach).
rt() | right()	Obraca żółwia w prawą stronę o podany w argumencie kąt (w stopniach).
pu() | penup() | up()	Podnosi pisak – do momentu opuszczenia pisaka żółw porusza się bez pozostawiania śladów.
pd() | pendown() | down()	Opuszcza pisak – żółw do momentu podniesienia pisaka będzie pozostawiał ślady.
goto() | setpos() | setposition()	Przemieszcza żółwia do określonej przez współrzędne x i y pozycji. Domyślnie na starcie żółw znajduje się na środku okna graficznego. Środek ten jest identyfikowany jako pozycja 0, 0. Współrzędna x rośnie w prawą stronę i maleje w lewą, a y rośnie w górę, maleje zaś w dół.

1 By skorzystać z modułu, należy go importować, co można zrobić, pisząc **import turtle**.

2 Już po imporcie modułu, by okno pojawiło się w programie, wystarczy użyć jednego z poleceń sterujących żółwiem. Jakie są to polecenia?
Przede wszystkim polecenie **fd** czy jego dłuższa forma – **forward** – pozwala przesunąć żółwia do przodu o określoną liczbę kroków. Domyślnie żółw jest zwrócony w prawą stronę i jeśli nie zmienimy tego, tylko od razu go przesuniemy, wykona ruch właśnie w prawą stronę.
Przez polecenia **right** i **left** możemy obrócić żółwia odpowiednio w kierunku prawym lub lewym – o określony kąt. Podstawowe polecenia dotyczące sterowania żółwiem są w tabeli obok.

PODSTAWY MODUŁU TURTLE – JAK WYRYSOWAĆ KWADRAT

By poznać podstawy obsługi modułu **turtle**, napiszemy skrypt tworzenia kwadratu.

1 Po imporcie modułu umieszczamy w skrypcie polecenie **turtle.fd()**, podając w nawiasie liczbę kroków, która jednocześnie będzie długością boku kwadratu.

```
*nowy żółw.py - C:/Users/konra/AppData/Local/
File  Edit  Format  Run  Options  Window  Help
import turtle

turtle.fd(100)
```

2 Ponieważ moduł turtle jest oparty na module tkinter, na końcu uruchamiamy metodę mainloop, pisząc **turtle.mainloop()**.

```
*nowy żółw.py - C:/Users/konra/AppData/L
File  Edit  Format  Run  Options  Window
import turtle

turtle.fd(100)

turtle.mainloop()
```

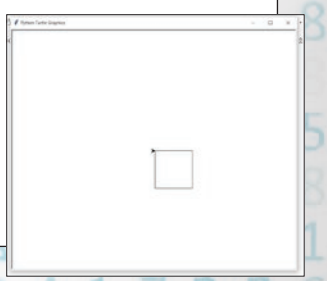
3 Już tak niewielki skrypt sprawi, że w oknie programu po jego uruchomieniu pojawi się pierwsza linia.

4 Drugi z boków kwadratu powinien być ustawiony prostopadle do pierwszego. To oznacza, że żółw przed narysowaniem tego boku powinien zostać wcześniej obrócony tak, by był skierowany w górę lub w dół. Jeśli chcemy skierować żółwia w dół, to względem jego obecnego ustawienia powinniśmy obrócić go w prawą stronę o 90 stopni, zatem należy dopisać polecenie **turtle.right(90)**.

```
File  Edit  Format  Run  Options  Window
import turtle

turtle.fd(100)
turtle.right(90)

turtle.mainloop()
```

5 Dla obróconego żółwia ponownie należałoby wykonać instrukcję przesuwającą go do przodu, by powstał kolejny bok kwadratu. Analizując kolejne posunięcia żółwia, zauważymy, że powinien on kolejno cztery razy przesunąć się i obrócić. Zamiast pisać kolejne instrukcje przesuwające i obracające, możemy skorzystać z prostszego sposobu polegającego na zapętleniu już zapisanych dwóch poleceń. Można to zrobić, korzystając z poznanej w poprzednim odcinku kursu pętli **for**. Zapisujemy przed instrukcjami do zapętlenia **for i in range(4):** i dodajemy wcięcia przed instrukcjami wymagającymi zapętlenia.

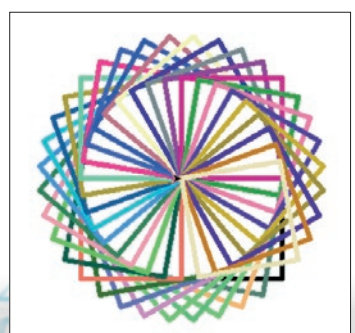
```
import turtle

for i in range(4):
    turtle.fd(100)
    turtle.right(90)

turtle.mainloop()
```



6 Po dodaniu pętli żółw będzie już wyrysowywał w oknie kwadrat.

ODWZOROWANIE RYSUNKU

Wiele zadań dotyczących wykorzystania modułu turtle polega na pisaniu takich skryptów, by odtworzyć w oknie programu założony rysunek. Takie zadanie wykonamy także w tym odcinku kursu. Przyjrzyjmy się rysunkowi obok. Przedstawia on 36 kwadratów w różnych kolorach, ustawionych w taki sposób, że ich rogi stykają się w środku okna programu.



1 Pierwszym krokiem realizacji tego zadania powinno być importowanie modułów. Będziemy potrzebować nie tylko modułu turtle, ale również modułu **random**, ponieważ będziemy musieli wygenerować liczby, które pozwolą na utworzenie różnych kolorów dla kwadratów.

```
File  Edit  Format  Run  Options  Wi
import turtle
import random
```

2 Dzięki poprzedniemu zadaniu wiemy, jak narysować jeden kwadrat. Podobna pętla **for** wykonująca się 4 razy powinna znaleźć się również w nowym kodzie źródłowym. Kwadrat powinien być narysowany jednak nie tylko jeden raz, ale 36 razy. Dlatego pętla rysująca kwadrat powinna być wewnątrz innej pętli, która wykona ją 36 razy.

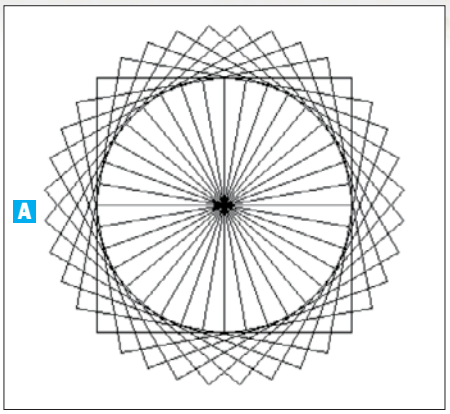
```
import turtle
import random

for j in range(36):
    for i in range(4):
        turtle.fd(100)
        turtle.right(90)
```

3 Zapętlenie rysowania kwadratu sprawi, że dostaniemy 36 kwadratów, z których wszystkie znajdują się w tym samym miejscu – jeden na drugim. By każdy kwadrat był nieco inaczej ustawiony, a co za tym idzie stał się widoczny po narysowaniu, po pętli tworzącej pojedynczy kwadrat, ale wewnątrz pęt li powtarzającej rysowanie kolejnych kwadratów, dodajemy polecenie obracające żółwia o 10 stopni w prawą stronę – **turtle.right(10)**.

```
for j in range(36):
    for i in range(4):
        turtle.fd(100)
        turtle.right(90)
    turtle.right(10)
```

4 Uzyskany w ten sposób efekt **A** nie jest jeszcze taki, jak na początkowej ilustracji – jednak samo położenie kwadratów pokrywa się już z naszymi oczekiwaniami.

5 Jedną z różnic pomiędzy oczekiwanym a uzyskanym efektem jest grubość linii, jaką rysowane są kształty. Zapiszmy więc **turtle.pensize(5)**, by zwiększyć jej grubość. Polecenie może znaleźć się na początku



skryptu, czyli po imporcie modułów, ale przed pętlami.

```
import turtle
import random

turtle.pensize(5)
for j in range(36
    for i in rang
```

6 Ostatnim elementem, który powinniśmy zmienić w naszym programie, jest generowanie nowych, różnych kolorów rysowania kwadratów. Kolory możemy określać zarówno poprzez nazwy, jak i w modelu RGB, gdzie każdy kolor przedstawiony jest jako połączenie czerwonego, zielonego i niebieskiego – każdej z tych trzech składowych odpowiada liczba od 0 do 255 mówiąca o tym, jak dużo danej barwy chcemy wziąć do utworzenia wynikowego koloru. By móc określać kolory w takich wartościach, należy ustawić colormode. Zapiszmy jeszcze przed pętlą polecenie **turtle.colormode(255)**.

```
turtle.pensize(5)
turtle.colormode(255)
for j in range(36):
    for i in range(4):
```

Prędkość rysowania

Żółw jako zwierzę nie jest demonem prędkości. Podobnie jest z naszym „żółwiem" rysującym w oknie graficznym. Mamy jednak dostęp do polecenia, które może przyspieszyć rysowanie. To polecenie to **turtle.speed()**, gdzie w nawiasie

```
import turtle
import random
turtle.speed(0)
turtle.pensize(5)
turtle.colormode(255)
for j in range(36):
```

podajemy oczekiwaną prędkość żółwia podczas rysowania. Prędkość ta powinna być wyrażona liczbą od 1 do 10, gdzie 1 oznacza wolnego żółwia, a 10 szybkiego żółwia. Do dyspozycji mamy też dodatkową opcję 0 – czyli najszybszą z możliwych prędkość żółwia.

7 Kolor możemy zmieniać już wewnątrz pętli – po narysowaniu każdego kwadratu. Kolor zapiszemy w zmiennej jako połączenie trzech wartości wygenerowanych przez polecenie **random.randrange(0,255)**, które pochodzi z modułu **random** i pozwala na wygenerowanie jednej liczby od 0 do 255. Cała potrzebna nam linia kodu powinna mieć postać: **kolor = (random.randrange(0,255), random.randrange(0,255), random.randrange(0,255))** **B**.

8 Wygenerowany kolor jest zapisywany do zmiennej kolor, jednak nie jest jeszcze przypisany jako kolor rysowania. By tak się stało, dopiszmy polecenie **turtle.pencolor(kolor)** **C** – również w pętli, po wygenerowaniu nowego koloru.

9 Po wprowadzeniu takich zmian program wyrysuje grafikę jak na ilustracji na poprzedniej stronie pokazującej efekt do osiągnięcia.

```
for j in range(36):
    for i in range(4):
        turtle.fd(100)
        turtle.right(90)
    turtle.right(10)
B   kolor = (random.randrange(0,255), random.randrange(0,255), random.randrange(0,255))
    kolor = (random.randrange(0,255), random.randrange(0,255), random.randrange(0,255))
C   turtle.pencolor(kolor)
turtle.mainloop()
```

MODUŁ TURTLE: GRA WYŚCIGI ŻÓŁWI

Moduł turtle można wykorzystać nie tylko do zadań polegających na odwzorowywaniu grafik. Można go użyć również do tworzenia gier i właśnie takie zastosowanie turtle poznamy w kolejnej wskazówce w tej części odcinka naszego kursu.

OKNO GRY I DWA ŻÓŁWIE

Zobaczmy, jak krok po kroku napisać grę **Wyścigi żółwi**. Gra przeznaczona jest dla dwóch graczy korzystających z tego samego komputera. Każdy z graczy będzie sterował jednym żółwiem. Obydwa żółwie będą mogły poruszać się tylko w jednym kierunku.

Wyrwij, użyj dziurkacza i zbieraj do segregatora

Każdy z graczy, wciskając inny klawisz na klawiaturze, przesuwa o pewną liczbę kroków swojego żółwia. Wygrywa ten, którego żółw pierwszy przekroczy linię mety, a inaczej mówiąc, wygra ten gracz, który będzie szybciej wciskał klawisz przesuwający żółwia do przodu.

1 Pierwszy etap tworzenia to import modułu turtle i wywołanie metody **mainloop**, które to do końca tworzenia kodu powinno pozostać na samym jego końcu A. Między tymi dwoma znanymi już poleceniami możemy umieścić jeszcze polecenie **turtle.bgcolor("pink")**, które pozwala na zmianę koloru tła w oknie na kolor podany w nawiasie – kolory możemy określać także poprzez ich angielskie nazwy.

```
*klawiatura z zolwiem.py - C:\Users\konra\AppData\Loca
File  Edit  Format  Run  Options  Window  Help
import turtle

turtle.bgcolor("pink")

turtle.mainloop()A
```

2 W naszej grze potrzebujemy dwóch żółwi. Gdy chcieliśmy sterować standardowym żółwiem, polecenia sterujące wywoływane były ze słowem **turtle** na początku. Możemy jednak tworzyć w kodzie, a co za tym idzie w oknie programu, więcej żółwi. Żeby utworzyć nową instancję żółwia, zapisujemy **t1 = turtle.Turtle()** – dzięki temu powstanie żółw o nazwie **t1**.

```
turtle.bgcolor("pink")

t1 = turtle.Turtle()
```

3 Każdy żółw jest domyślnie widoczny na środku okna. Jeśli tworzymy wyścigi – powinnyśmy widzieć obydwa ścigające się żółwie. Dlatego żółw t1 powinien być przesunięty na starcie programu. Przed przesunięciem żółwia należy napisać **t1.pu()**, by żółw nie zostawiał śladu od środka okna do miejsca docelowego jego przesunięcia. Przesunięcie do miejsca docelowego uzyskamy, pisząc **t1.setpos(-150, 100)**. Żółw w ten sposób przesunie się o 150 kroków w lewą stronę względem środka i o 100 kroków w górę.

```
t1 = turtle.Turtle()
t1.pu()
t1.setpos(-150, 100)
```

4 Każdy żółw powinien mieć inny kolor. By zmienić kolor żółwia t1 na żółty, piszemy **t1.pencolor("yellow")**.

```
t1.pu()
t1.setpos(-150, 100)
t1.pencolor("yellow")
```

5 Standardowa grafika żółwia jest niewielka. Możemy ją jednak powiększyć, by lepiej prezentowała się w oknie gry. By to zrobić, piszemy **t1.shapesize(5, 5, 12)**. Pierwsza liczba w nawiasie oznacza przeskalowanie

```
t1.pencolor("yellow")
t1.shapesize(5, 5, 12)
```

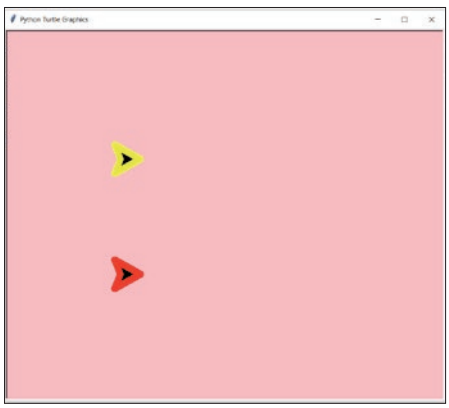
żółwia w poziomie, a druga w pionie. Zatem żółw urośnie pięciokrotnie. Ostatnia z liczb w nawiasie to grubość linii, jaką narysowana jest obwódka żółwia – dzięki niej będziemy lepiej odróżniać żółwie w oknie gry.

6 By utworzyć drugiego żółwia – o nazwie **t2** – należy wykonać ten sam zestaw instrukcji co w przypadku żółwia t1. Żółwie różnią się między sobą nie tylko nazwą, ale także kolorem – drugi żółw ma kolor czerwony – oraz pozycją, bo drugi z żółwi jest

```
t1 = turtle.Turtle()
t1.pu()
t1.setpos(-150, 100)
t1.pencolor("yellow")
t1.shapesize(5, 5, 12)

t2 = turtle.Turtle()
t2.pu()
t2.setpos(-150, -100)
t2.pencolor("red")
t2.shapesize(5, 5, 12)
```

przesunięty o 100 kroków w dół w stosunku do środka okna.

7 Na tym etapie tworzenia gry mamy już okno z dwoma żółwiami.



8 Żółwie zwrócone są w prawą stronę, zatem po ich prawej stronie powinna pojawić się linia pełniąca funkcję mety, do której żółwie będą musiały przejść. By utworzyć metę, możemy wygenerować nowego żółwia o nazwie **meta**, który ją narysuje.

```
t2.pencolor("red")
t2.shapesize(5, 5, 12)

meta = turtle.Turtle()
```

9 Nowy żółw standardowo jest na środku okna – dlatego trzeba go przesunąć w prawą stronę. Jednak zanim to zrobimy, trzeba użyć polecenia **meta.pu()** B, by żółw podczas przesuwania nie zostawiał śladów. Żółwia przesuwamy 250 kroków w prawą stronę względem środka i 150 kroków w górę.

```
meta = turtle.Turtle()
meta.pu()B
meta.setpos(250, 150)
```

10 Żółwie mogą umieszczać w oknach graficznych napisy. Służy do tego metoda **write**, która użyta w formie **meta.write("META")** sprawi, że w oknie wstawiony będzie napis **META**.

```
meta.setpos(250, 150)
meta.write("META")
```

11 Oprócz napisu meta powinna być utworzona linia. Linia ta może być nieco grubsza niż standardowa grubość linii domyślnie rysowanej przez żółwia. Zapiszmy **meta.pensize(5)**. Pamiętajmy jednak, że

```
meta.write("META")
meta.pensize(5)
```

na razie żółw, poruszając się, nie pozostawia śladów. Dlatego należy użyć polecenia **meta.pd()** C, by żółw „przyłożył pisak" do powierzchni.

```
meta.pensize(5)
meta.pd()C
meta.right(90)D
meta.fd(300)E
```

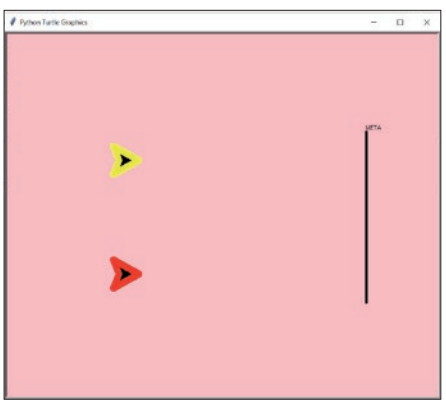
12 Linia mety powinna być pionowa. By ją stworzyć, należy obrócić żółwia względem obecnego kierunku, tak by patrzył w dół okna, czyli o 90 stopni w prawą stronę D.

13 Dopiero teraz można użyć polecenia **fd()** E, by przesunąć żółwia do przodu, i narysować w ten sposób metę o długości 300 kroków.

14 Po narysowaniu mety żółw, który ją stworzył, nie jest nam już potrzebny. Możemy go ukryć, pisząc **meta.hideturtle()**.

```
meta.fd(300)
meta.hideturtle()
```

15 W ten sposób mamy już przygotowane okno gry zawierające nie tylko żółwie, ale i linię mety.



REAKCJA NA WCIŚNIĘCIE KLAWISZA

Nasze żółwie powinny reagować na wciskanie klawiszy. Każdy z żółwi będzie obsługiwany oddzielną funkcją – a każda z tych funkcji będzie przypisana do wciśnięcia innego klawisza na klawiaturze. Funkcje te należy zdefiniować według przykładu.

1 Na początku każdej z funkcji przesuwamy żółwia o 10 kroków do przodu.

2 Potem przez instrukcję warunkową sprawdzamy, czy współrzędna **x** żółwia odczytana z funkcji **xcor** osiągnęła już taką wartość, że żółw dotarł do mety. I jednocześ nie, docierając do mety, żółw ma być wciąż widoczny.

3 Jeśli tak jest, żółw wyświetla napis **WYGRYWAM** i obydwa żółwie, niezależnie od tego, który z nich dotarł do mety, zostają ukryte.

```
def up1():
    t1.forward(10)
    if t1.xcor() > 250 and t1.isvisible():
        t1.write("WYGRYWAM")
        t2.hideturtle()
        t1.hideturtle()


def up2():
    t2.forward(10)
    if t2.xcor() > 250 and t2.isvisible():
        t2.write("WYGRYWAM")
        t1.hideturtle()
        t2.hideturtle()
```

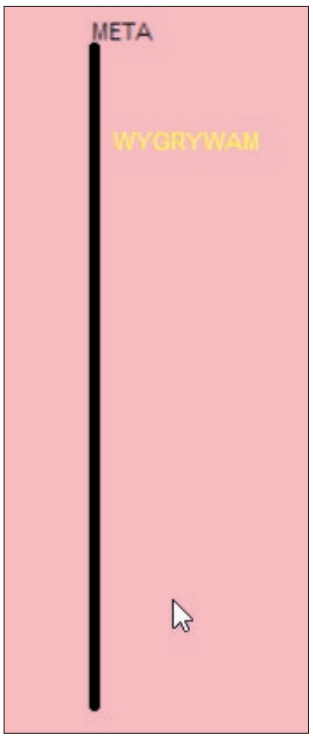
PRZYPISANIE FUNKCJI DO ZDARZENIA

1 Poprzez polecenie **turtle.onkey()** możemy przypisać wykonywanie konkretnej funkcji do wciśnięcia podanego klawisza na klawiaturze. I tak zapis **turtle.onkey(up1,'Up')** powoduje wywoływanie funkcji **up1** zawsze, gdy wciskana jest strzałka w górę, a **turtle.onkey(up2,'w')** pozwala na wywoływanie funkcji **up2** przesuwającej czerwonego żółwia, gdy wciskamy klawisz W.

```
t2.hideturtle()

turtle.onkey(up1,'Up')
turtle.onkey(up2,'w')
turtle.listen()
turtle.mainloop()
```

2 By program nasłuchiwał wciskania klawiszy na klawiaturze, należy dopisać jeszcze polecenie **turtle.listen()**.

Przetestujmy grę, aby sprawdzić, czy gdy jeden z żółwi dotrze do mety, to obydwa żółwie znikają, a w miejscu żółwia, który wygrał, pojawia się informacja o wygranej.



MODUŁ TURTLE: GRA UNIKAJ LINII

Kolejnym projektem, jaki omówimy w tym odcinku kursu, jest gra, w której żółw porusza się po oknie, nie mogąc się zatrzymać. Gracz za pomocą strzałek na klawiaturze może jedynie zmieniać kierunek, w którym podąża żółw.

Zadaniem gracza jest takie sterowanie żółwiem, by ten nie wszedł na pozostawiany przez siebie ślad, ale też nie opuścił okna gry. Im dłużej gracz utrzyma żółwia w ruchu, tym więcej zdobywa punktów. Zobaczmy, jak przygotować taką grę.

GRA UNIKAJ LINII – KROK PO KROKU

1 Skrypt gry rozpoczynamy od importowania modułu **turtle**.

2 Już na początku możemy umieścić w skrypcie także linię **turtle.mainloop()**, która powinna znajdować się na końcu kodu gry.

3 Po imporcie modułu można utworzyć obiekt o nazwie **gracz**, który będzie naszym nowym żółwiem.

```
import turtle

gracz= turtle.Turtle()
```

4 Na tym etapie można określić, z jaką prędkością żółw będzie poruszał się po oknie gry. Wywołując metodę **gracz.speed()**, możemy określić prędkość rysowania. W tej grze potrzebujemy dość szybkiego żółwia, aby była ona grywalna. Stąd ustawiamy **gracz.speed** na **0** – to maksymalna wartość prędkości.

```
gracz= turtle.Turtle()
gracz.speed(0)
```

5 Do zmiany kierunku żółwia w tej grze nie będziemy wykorzystywać obrotów o określony kąt. Zakładamy, że żółw może poruszać się tylko w stronę prawą, lewą, górę lub dół – czyli w jednym z czterech zdefiniowanych kierunków. Do ustawienia kierunku służy polecenie **setheading** wywoływane dla konkretnego żółwia w oknie. W trybie standardowym domyślny kierunek, w jakim zwrócony jest żółw na starcie programu, ma wartość 0 – co oznacza prawą stronę. Zatem polecenie **gracz.setheading(0)** równo-

```
def Right(): A
    gracz.setheading(0)
```

znaczne jest z ustawieniem kierunku żółwia na prawą stronę. By taki kierunek żółw przybrał po wciśnięciu strzałki w prawo, powinno ono znaleźć się w funkcji, która dalej będzie przypisana do zdarzenia wciśnięcia tego klawisza. Zdefiniujmy zatem funkcję – może ona mieć nazwę **Right** A – i umieśćmy w niej omawiane polecenie.

6 Analogicznie będziemy potrzebować podobnych funkcji dla zmiany kierunku na stronę lewą, górę i dół. Zdefiniujmy zatem

Wyrwij, użyj dziurkacza i zbieraj do segregatora

trzy kolejne funkcje – **Left** B, **Up** C i **Down** D. Każda z nich powinna wykonywać polecenie **gracz.setheading**, jednak w każdej z nich powinien być inny docelowy kierunek ustawienia żółwia. Kolejne kierunki możemy wyliczyć, przyjmując, że są one podawane jako miara kąta obrotu z ustawienia początkowego w kierunku przeciwnym do ruchu wskazówek zegara. Zatem żółw z kierunkiem 90 jest zwrócony w górę – stąd **gracz.setheading(90)** E powinno znaleźć się w funkcji **Up** ustawiającej kierunek na górę. Polecenie **gracz.setheading(180)** F powinno znaleźć się w funkcji **Left**, a **gracz.setheading(270)** G powinno znaleźć się w funkcji **Down**.

```
def Right():
    gracz.setheading(0)

def Left():B
    gracz.setheading(180)F

def Up():C
    gracz.setheading(90)E

def Down():D
    gracz.setheading(270)G
```

7 By każda ze zdefiniowanych funkcji wykonywała się podczas kliknięcia strzałek, należy każdą z nich oddzielnie przypisać do wciśnięcia odpowiadającej jej strzałki poprzez polecenie **turtle.onkey**. Nazwy klawiszy określane w poleceniu są takie same, jak użyte w przykładzie nazwy funkcji.

```
    gracz.setheading(270)

turtle.onkey(Right,'Right')
turtle.onkey(Left,'Left')
turtle.onkey(Up,'Up')
turtle.onkey(Down,'Down')
```

8 By uruchomić tryb nasłuchiwania wciśnięcia klawiszy na klawiaturze, należy wywołać jeszcze metodę **turtle.listen()**.

```
turtle.onkey(Down,'Down')

turtle.listen()
```

9 Żółw, poruszając się po oknie, będzie pozostawiał po sobie ślady, na które nie będzie można wejść ponownie – ponieważ będzie to skutkowało zatrzymaniem gry. By móc sprawdzać, czy żółw jest w pozycji, w której się już kiedyś znajdował, stworzymy listę odwiedzanych przez niego pozycji. By utworzyć listę, podajemy jej nazwę – w tym przypadku będzie to **odwiedzone** i jako wartość przypiszemy jej pusty nawias kwadratowy.

```
turtle.listen()
odwiedzone = []
```

10 Zdefiniujemy też funkcję **ruch**, która będzie odpowiadała za dodawanie kolejnych pozycji, w jakich znajduje się żółw do listy odwiedzonych, ale też przesuwała samego żółwia w określonym przez strzałki kierunku. I właśnie przesunięcie żółwia – czyli wywołanie **gracz.fd()** H – powinno znaleźć się na początku tej funkcji. W nawiasie podajemy, ile kroków w każdym ruchu ma pokonać żółw – na przykład 10. Im większą liczbę kroków podamy, tym trudniejsza będzie gra.

```
odwiedzone = []

def ruch():
    gracz.fd(10)
H
```

11 Oprócz przesuwania żółwia powinniśmy sprawdzać, czy pozycja, w której się on znajduje (odczytujemy ją z metody **gracz.pos()**), jest na liście odwiedzonych pozycji. Robimy to, umieszczając w instrukcji warunkowej warunek **gracz.pos() i odwiedzone**.

```
def ruch():
    gracz.fd(10)
    if gracz.pos() in odwiedzone:
        gracz.hideturtle()I
```

12 Jeśli warunek jest spełniony, możemy użyć polecenia **gracz.hideturtle** I, by ukryć żółwia – a jego ukrycie będzie równoznaczne z końcem gry.

13 By zaś sama funkcja mogła się wykonywać, możemy po prostu wywołać ją jednokrotnie – i kolejne jej wywołania umieścić wewnątrz tej samej funkcji – lub użyć polecenia **turtle.ontimer(ruch, 1)**, gdzie **ruch** jest nazwą funkcji, a **1** liczbą jej kolejnych wywołań.

```
turtle.ontimer(ruch, 1)
turtle.mainloop()
```

14 Kolejne wywołania funkcji powinny następować tylko wtedy, kiedy pozycji, w jakiej jest żółw, nie ma na liście odwiedzonych pozycji. By cokolwiek wykonywało się, tylko gdy żółw jest na nieodwiedzonej pozycji, do utworzonej instrukcji warunkowej dodajemy sekcję **else**, pozwalającą na określenie, co ma się wykonać, gdy warunek nie jest spełniony.

```
if gracz.pos() in odwiedzone:
    gracz.hideturtle()
else:
    odwiedzone.append(gracz.pos())J
```

15 Poprzez polecenie append można dodawać do list nowe elementy. Zapisanie **odwiedzone.append(gracz.pos())** J sprawi, że pozycja, w której jest obecnie żółw, zostanie dodana do listy odwiedzonych pozycji.

16 Również w sekcji **else** instrukcji warunkowej ponownie użyjemy **turtle.ontimer(ruch, 1)**, by ponownie wykonać funkcję przesuwającą żółwia.

```
else:
    odwiedzone.append(gracz.pos())
    turtle.ontimer(ruch, 1)
```

17 Uruchamiając grę w tym momencie, zobaczymy, że możemy już sterować żółwiem. Gdy wejdzie on na pozostawioną wcześniej przez siebie linię, sama grafika „strzałki" symbolizującej żółwia znika, ale linie nadal są rysowane. Będziemy musieli jeszcze nieco rozbudować skrypt, by żółw oprócz zniknięcia przestawał w tym momencie także rysować linię. Po istniejącej już w funkcji **ruch** instrukcji warunkowej dodajemy kolejną – w niej umieszczamy warunek **gracz.isvisible() == False**, co pozwoli nam na dodanie kolejnych poleceń, które wykonają się wtedy, gdy żółw będzie już niewidoczny.

```
    odwiedzone.append(gracz.pos())
    turtle.ontimer(ruch, 1)
if gracz.isvisible() == False:
```

18 Co wtedy powinno stać się z żółwiem? Powinien przestać rysować. Ten efekt odniesiemy, wywołując **gracz.pu()**.

```
if gracz.isvisible() == False:
    gracz.pu()
```

19 Użytkownik naszej gry powinien wiedzieć, że gra się zakończyła. Dlatego najpierw przenosimy żółwia do pozycji startowej, na środek okna gry – poleceniem **gracz.home()**, a potem poleceniem **gracz.**

```
if gracz.isvisible() == False:
    gracz.pu()
    gracz.home()
```

write("KONIEC GRY :(") umieszczamy tam napis informujący o zakończeniu rozgrywki.

```
if gracz.isvisible() == False:
    gracz.pu()
    gracz.home()
    gracz.write("KONIEC GRY :(")
```

20 Rozgrywka powinna kończyć się nie tylko, gdy gracz wejdzie na swój ślad, ale też, gdy żółw będzie próbował opuścić okno gry. Instrukcję warunkową sprawdzającą, czy pozycja, w której jest żółw, znajduje się na liście odwiedzonych pozycji, należy rozbudować o sprawdzenie kolejnych warunków, których spełnienie będzie powodowało ukrycie żółwia. Kolejne warunki do sprawdzenia umieszczamy przed sekcją **else**, dopisując do instrukcji sekcję **elif**. O ile w jednej instrukcji warunkowej może być tylko jedna sekcja **else**, o tyle sekcji **elif** może być wiele, a każda może sprawdzać inny warunek. W pierwszej sekcji **elif** sprawdzimy, czy żółw doszedł do górnej krawędzi okna. W tym celu odczytujemy współrzędną y żółwia – pisząc **gracz.ycor()** K, i sprawdzamy, czy jest ona większa niż współrzędna, jaką miałby żółw, będąc przy górnej krawędzi – a miałby tam współrzędną odpowiadającą połowie wysokości okna gry – samą wysokość okna odczytujemy, wywołując **turtle.window_height()** L.

```
if gracz.pos() in odwiedzone:
    gracz.hideturtle()
elif gracz.ycor() > turtle.window_height() / 2:L
  K gracz.hideturtle()
```

21 Kolejna sekcja **elif** powinna sprawdzać, czy żółw dotarł do dolnej krawędzi okna gry. Współrzędna y żółwia przesuwającego się do dołu zmniejsza się i w dolnej części okna ma wartości ujemne. Jeśli współrzędna y ma wartość mniejszą niż wyrażona

```
elif gracz.ycor() > turtle.window_height() / 2:
    gracz.hideturtle()
elif gracz.ycor() < -turtle.window_height() / 2:
    gracz.hideturtle()
```

wartością ujemną połowa wysokości okna gry M, także powinniśmy ukryć żółwia.

22 Analogicznie powinniśmy dodać dwie kolejne sekcje **elif** – sprawdzające, czy żółw wyszedł z okna gry prawą lub lewą krawędzią. Jednak nie uwzględniamy już współrzędnej y, lecz współrzędną x – a także nie wysokość, ale szerokość okna, którą to odczytujemy z **turtle.window_width()**.

```
elif gracz.ycor() < -turtle.window_height() / 2:
    gracz.hideturtle()
elif gracz.xcor() > turtle.window_width() / 2:
    gracz.hideturtle()
elif gracz.xcor() < -turtle.window_width() / 2:
    gracz.hideturtle()
else:
```

23 Gra już działa i zatrzymuje się w odpowiednich momentach, jednak nie możemy sprawdzić uzyskanego przez gracza wyniku. By było to możliwe, zadeklarujmy przed funkcją **ruch** zmienną o nazwie **punkty** i wartości początkowej równej **0**.

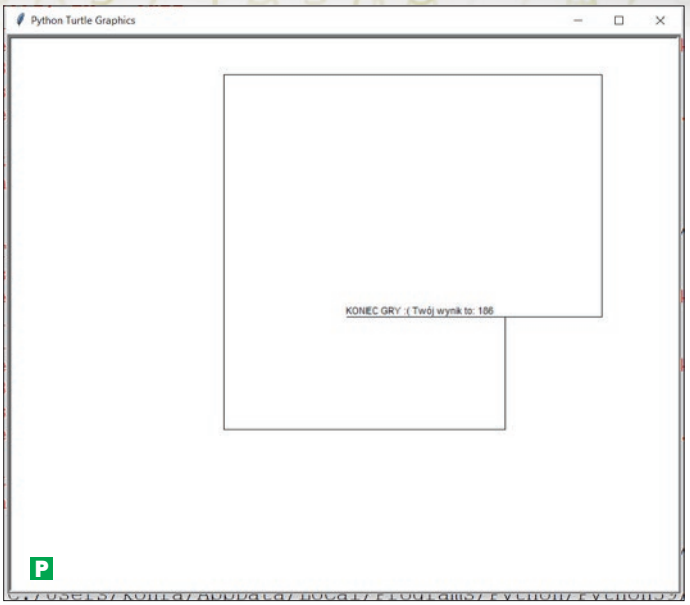
```
punkty = 0
def ruch():
    gracz.fd
```

24 Wewnątrz funkcji **ruch** z każdym jej wywołaniem należy zwiększać wartość zmiennej **punkty** o **1** N. Jednak by punkty w obrębie funkcji i **punkty** poza funkcją wskazywały na tę samą wartość, należy przed użyciem zmiennej **punkty** w funkcji napisać **global punkt**, inaczej program będzie zwracał błędy.

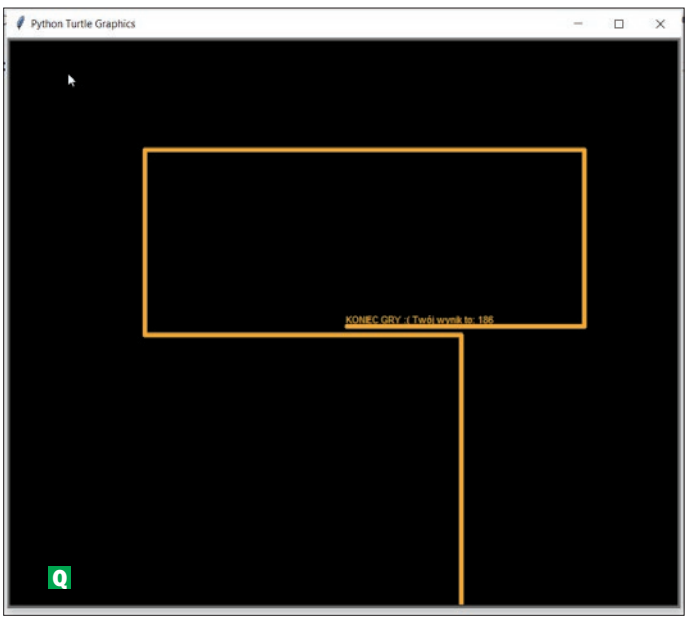
```
def ruch():
    gracz.fd(10)
    punkty = punkty + 1 N
```

```
    gracz.fd(10)
    global punkty
    punkty = punkty + 1
```

25 Gdy punkty są obliczane, gracz powinien być informowany o tym, jaka wartość punktów została zmagazynowana – dlatego można zmodyfikować polecenie umieszczające komunikat o końcu gry w oknie gry, tak by dodatkowo wypisana została także wartość zmiennej **punkty** O.



26 Gra już działa – poruszając się po oknie gry, żółw zostawia ślady P. Jeśli na nie wejdziemy, gra zostaje zakończona, a gracz może sprawdzić, ile zdobył punktów.



27 Jeśli chcemy nieco poprawić wygląd gry – możemy na początku skryptu użyć kilku poleceń zmieniających wygląd – **gracz.pencolor** pozwoli zmienić kolor linii, **gracz.pensize**jej grubość, a **turtle.bgcolor** kolor tła. Po zmianach okno gry Q może prezentować się nieco lepiej.

```
gracz.home()
gracz.write("KONIEC GRY :( Twój wynik to: "+str(punkty))
```
O

Co dalej

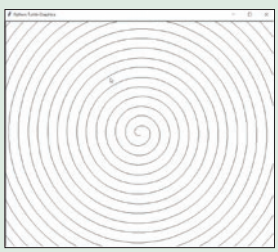
Kolejna część kursu tematycznie będzie się łączyć z tą częścią, ponieważ będzie dotyczyć modułu **pygame**. W module tym zawarte są zasoby ułatwiające tworzenie gier. Takim zasobem jest także okno graficzne, w którym możemy wyświetlać gry.

ZADANIA DO SAMODZIELNEGO WYKONANIA

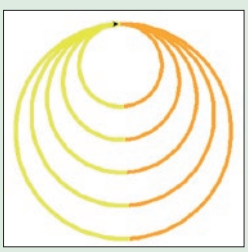
SKRYPTY DO POBRANIA Z KSPLUS.PL

Jak już wiemy z tego odcinka kursu, moduł **turtle** jest wykorzystywany do nauki programowania poprzez rozwiązywanie zadań polegających na pisaniu takich kodów źródłowych, których działanie ma odwzorowywać założone w zadaniu obrazy. Rozwiąż teraz samodzielnie kilka takich zadań. **Uwaga!** Skrypty z rozwiązaniami tych zadań znajdziemy w KŚ+ (ksplus.pl).

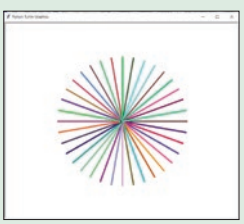
■ **Zadanie 1** Napisz skrypt, w którym żółw narysuje spiralę, taką jak w przykładowym oknie graficznym.



■ **Zadanie 2** W ramach kolejnego zadania, napisz program, w którym żółw wyrysuje pięć dwukolorowych kół, jak na przykładowej ilustracji.



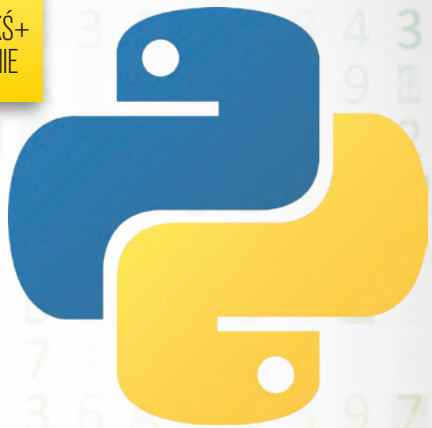
■ **Zadanie 3** To zadanie polega na ułożeniu z kolorowych kresek zaprezentowanego na ilustracji koła. Przed rysowaniem każdej kreski powinniśmy na nowo generować kolor rysowania.



Wyrwij, użyj dziurkacza i zbieraj do segregatora

TWORZENIE GIER Z WYKORZYSTANIEM MODUŁU PYGAME

POPRZEDNIE ODCINKI ZNAJDZIESZ W KŚ+ JAKO E-WYDANIE



Python to język o wielu zastosowaniach. By stworzyć program o konkretnych funkcjach, wykorzystuje się często rozmaite moduły ułatwiające różne zadania.

Tak jest również z tworzeniem gier – aby pisać je w Pythonie, możemy zastosować gotowe moduły, które takie zadanie bardzo nam ułatwią.

Jednym z takich modułów jest **pygame**. Z tego odcinka kursu programowania w języku Python dowiemy się, jak posługiwać się tym modułem.

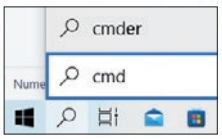
MODUŁ DO TWORZENIA GIER

Znamy już podstawy korzystania z modułów **tkinter** i **turtle** do tworzenia programów mających okno graficzne. Zarówno wygląd okien, jak i podejście do ich stworzenia i wykorzystywane przy tym polecenia różnią się w zależności od tego, na jakim module opieramy nasz skrypt. Tkinter wykorzystywany jest głównie do pisania oprogramowania użytkowego, a turtle – do zadań związanych z edukacją. Ale oba gorzej sprawdzają się przy tworzeniu gier.

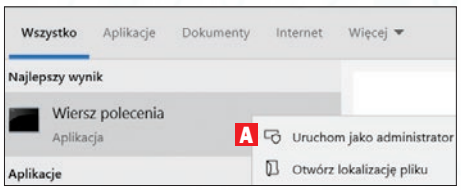
Dlatego, jeśli myślimy o stosowaniu Pythona do tworzenia gier, warto poznać moduł, który właśnie w tym celu został stworzony. Mowa tu o module **pygame**. Warto jednak pamiętać, że jest to tylko jeden z gamy modułów, jakie można wykorzystać przy tworzeniu gier.

Tego typu moduły nie są standardowo dołączone do Pythona i wymagają instalacji, aby można było programować za ich pomocą. Do instalacji modułów służy narzędzie pip obsługiwane z poziomu Wiersza polecenia.

1 By uruchomić Wiersz polecenia, w polu wyszukiwania na pasku menu **Start** wpisujemy **cmd**.

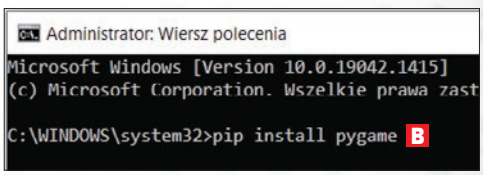


2 Z listy wyszukanych opcji wybieramy pozycję **Wiersz polecenia** i klikamy na nią prawym przyciskiem myszy, by wyświetlić menu kontekstowe.



3 Z niego wybieramy opcję **Uruchom jako administrator** A – te uprawnienia mogą być potrzebne do przeprowadzenia instalacji.

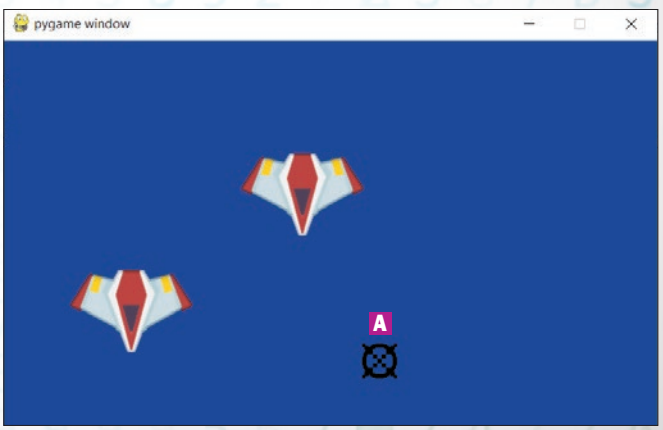
4 W otwartym oknie Wiersza polecenia wpisujemy **pip install pygame** B i wciskamy enter, by uruchomić podane polecenie. Po chwili w oknie powinien pojawić się komunikat o instalacji potrzebnego w tym rozdziale modułu pygame.



GRA – ZBIJANIE STATKÓW KOSMICZNYCH

Zakładamy, że chcemy stworzyć grę, w której zadaniem gracza będzie trafienie w jak największą liczbę statków kosmicznych.

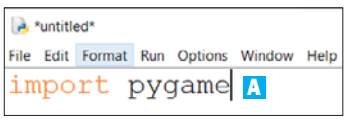
Statki będą przelatywać przez okno gry. Będą przemieszczać się od góry do dołu. Będziemy w nie trafiać po najechaniu i kliknięciu na statek kursorem myszy, który w obrębie okna naszej gry będzie miał kształt celownika A.



Fot. Photo5963/iStockphoto.com

OKNO GRY MODUŁU PYGAME

1 Na początek musimy napisać polecenie, którym zaimportujemy zainstalowany właśnie moduł do programu. Import dodatkowo instalowanych modułów nie różni się od importu modułów wbudowanych i możemy go przeprowadzić na dwa znane już z tego kursu sposoby – poleceniem **import pygame** A lub **from pygame import*** – w tym wypadku dalej w kodzie polecenia z modułu pisalibyśmy bez podawania jego nazwy. Nasza gra może jednak wymagać użycia więcej niż jednego modułu. W takich sytuacjach lepiej stosować pierwszy ze sposobów, ponieważ nazwy wykorzystywane w różnych modułach mogą się powielać.



2 Aby móc swobodnie wykorzystywać polecenia z modułu **pygame**, należy wywołać dla niego polecenie **init()** – poprzez linijkę **pygame.init()** B – jest to polecenie uruchamiające dla modułu.

```
import pygame

pygame.init()B
```

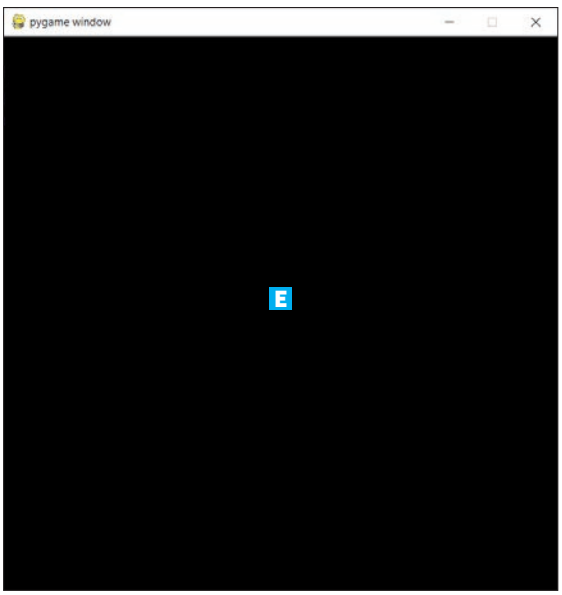
3 Ponieważ pygame podobnie jak tkinter daje nam do dyspozycji okno graficzne, jego utworzenie na potrzeby gry będzie kolejnym krokiem. Podczas tworzenia okna, korzystając z modułu pygame, podajemy wymiary. By nasza gra działała prawidłowo, dalej będziemy musieli stosować te same wymiary. Aby było nam wygodnie z nich korzystać, utwórzmy dwie zmienne do ich przechowywania – **szer** i **wys** C. To do nich wpisujemy odpowiednio szerokość i wysokość okna wyrażone w pikselach, jakie chcemy mieć w swojej grze. Może to być 600 i 600 pikseli, jak w przykładowym skrypcie.

```
pygame.init()
szer = 600
wys = 600
```
C

4 Mając zapisane wymiary okna, możemy przejść do użycia polecenia, które tworzy potrzebne nam do gry okno graficzne. Tak jak w przypadku tkintera, okno musi mieć swoją nazwę. Możemy je nazwać po prostu **okno** – a następnie, poprzez polecenie **pygame.display.set_mode((szer,wys))** D, przypisać do tej nazwy okno gry o wymiarach określonych wcześniej i zapisanych do zmiennych.

```
szer = 600
wys = 600
okno = pygame.display.set_mode((szer,wys))
```
D

5 Na tym etapie przy próbie uruchomienia skryptu możemy zobaczyć już utworzone okno E. Nie działa ono jednak perfekcyjnie.



Głównym elementem wymagającym poprawy jest obsługa przycisku, który je zamykaj.
Niestety, bez dodania odpowiedniej części skryptu okno utworzone poprzez moduł pygame nie daje się zamknąć inaczej niż poprzez wyjście z powłoki (**Shell**), czyli konsoli.

```
okno = pygame.display.set_mode((szer
wys = 600
okno = pyga
while True:
```
F G H

6 Niezbędne do zaprogramowania mechanizmu zamykania okna jest poznanie zagadnienia **eventów**, czyli tak zwanych **zdarzeń**.
Zdarzenia to różne akcje możliwe do wykonania przez użytkownika. Zdarzeniem jest właśnie kliknięcie na przycisk z iksem w rogu okna gry, ale też naciś nięcie jakiegokolwiek klawisza na klawiaturze czy kliknięcie myszą, a także zwolnienie przycisku czy klawisza.
Pygame ma listę eventów, które wystąpiły w danej chwili. Jeśli coś się zadziało, to trafia na listę, po czym za chwilę ją opuszcza, gdy przestaje się dziać.
Naszym zadaniem powinno być sprawdzanie tej listy w celu znalezienia zdarzenia, które mówi o próbie kliknięcia na przycisk z iksem w rogu okna. Lista zdarzeń zmienia zawartość – zdarzenia znajdują się na niej tylko w momencie występowania. Dlatego sprawdzanie zawartości listy powinno odbywać się możliwie często.
Aby powtarzać sprawdzanie, należy umieścić je w pętli. Możemy skorzystać z pętli **while** F, która może wykonywać coś, dopóki spełniony jest określony w niej warunek. Na potrzeby naszej gry w warunku możemy wpisać słowo **True** G oznaczające prawdę, a pętla będzie wykonywać się cały czas, aż do zamknięcia okna.

7 By odnaleźć interesujące nas zdarzenie, powinniśmy analizować każdy element listy zdarzeń i porównywać go ze zdarzeniem, którego szukamy. Do każdego ze zdarzeń możemy się odnosić poprzez odpowiednio skonstruowaną pętlę **for**. Lista z występującymi aktualnie zdarzeniami zwracana jest poleceniem **pygame.event.get()**, zatem pętla **for** może wyglądać następująco: **for event in pygame.event.get():** H. Pamiętajmy jednak o tym, by pętla ta znalaz ła się wewnątrz utworzonej w poprzednim kroku pętli **while**, którą będziemy nazywać **pętlą główną gry**.

8 Zgodnie z konstrukcją pętli pod nazwą **event** w każdym przejściu pętli będzie znajdować się kolejne zdarzenie z listy. Dalej należy sprawdzić – poprzez instrukcję warunkową – czy typ zdarzenia z listy zgadza się z tym, którego poszukujemy. Zapisujemy to linijką **if event.type == pygame.QUIT** I, ponieważ **pygame.QUIT** to typ zdarzenia wskazujący na naciśnięcie przycisku z iksem w rogu okna.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```
I

9 Zatem gdy zapisany warunek jest spełniony, należy wykonać polecenie – **pygame.quit()** J, co zamyka okno gry.

```
if event.type == pygame.QUIT:
    pygame.quit()
```
J

10 Dodajmy jeszcze polecenie **quit()** K – ono dodatkowo zakończy działanie całego programu, a nie tylko części odpowiedzialnej za okno.

```
if event.type == pygame.QUIT:
    pygame.quit()
```
K

11 Jak widzimy, w ten sposób powstał szkielet okna gry.
Zapisane do tej pory instrukcje mogą być wykorzystywane w wielu kolejnych projektach, jakie zechcemy stworzyć z wykorzystaniem modułu **pygame**.
Przetestujmy jeszcze działanie przycisku zamykającego okno.
Jeśli wszystko działa prawidłowo, możemy przejść do kolejnej części realizacji gry, w której opisane są już kroki odpowiedzialne za realizację mechanizmów specyficznych dla realizowanego właśnie projektu. Taki szkielet warto zapisać w oddzielnym pliku, by móc rozpoczynać od niego pracę nad kolejnymi projektami.

Wyrwij, użyj dziurkacza i zbieraj do segregatora

DODAJEMY CELOWNIK DO OKNA GRY

Celownik, który powinien pojawić się w oknie gry, będzie wczytywany z zewnętrznego pliku graficznego. Przygotujmy zatem grafikę z celownikiem A. Warto wykorzystać do tego plik w formacie PNG z przezroczystym tłem.

A

1 Gdy plik graficzny jest już zapisany na dysku w odpowiedniej lokalizacji, można przejść do zapisania odpowiednich instrukcji w kodzie, które pozwolą na wczytanie pliku do programu. By móc odnosić się do naszego celownika w kodzie, powinniśmy nadać mu nazwę – jak zmiennej. Zapisujemy w kodzie nazwę **celownik**. Wczytanie pliku graficznego nie musi odbywać się wielokrotnie. Raz wczytany plik możemy cały czas wykorzystywać w programie, bez konieczności ciągłego wczytywania. Dlatego celownik zaczniemy tworzyć przed pętlą główną gry. Do zapisanej nazwy należy wczytać teraz odpowiednią zawartość, co można zrobić poleceniem **pygame.image.load('celownik.png')** B, w którym **'celownik.png'** oznacza nazwę pliku graficznego wraz z jego rozszerzeniem.

```
okno = pygame.display.set_mode((szer,wys))

celownik = pygame.image.load('celownik.png')

while True:
```
B

2 Dodane polecenie pełni funkcję wczytania zasobu do projektu, a nie umieszczenia pliku w oknie. By plik znalazł się w oknie, trzeba użyć kolejnego polecenia. Zawartość okna będzie wraz z trwaniem rozgrywki się zmieniała. Za zmiany te będzie odpowiadać pętla główna gry. To właś nie w niej powinniśmy zapisać polecenie umieszczają-

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    okno.blit(celownik, (100, 150))
```
C

ce w oknie gry grafikę celownika. Wewnątrz pętli głównej zapisujemy **okno.blit(celownik, (100, 150))** C – gdzie okno to nazwa okna graficznego, której wcześniej użyliśmy w kodzie, **celownik** zaś to nazwa obiektu graficznego, wczytanego w poprzednim kroku.

3 Jak widzimy, w poleceniu tym znajdują się też liczby – 100 i 150. Możemy użyć innych liczb. Pierwsza z nich oznacza odległość od lewej krawędzi okna gry, w jakiej celownik zostanie umieszczony w oknie, a druga to odległość od górnej krawędzi okna gry. Wpisane w tym kroku liczby są tymczasowe – służą jedynie sprawdzeniu tego, czy celownik już pojawia się w oknie. Docelowo powinny być one zastąpione wartościami odczytywanymi z pozycji kursora, tak by celownik zmieniał swoją lokalizację w oknie gry wraz z ruchem myszy.

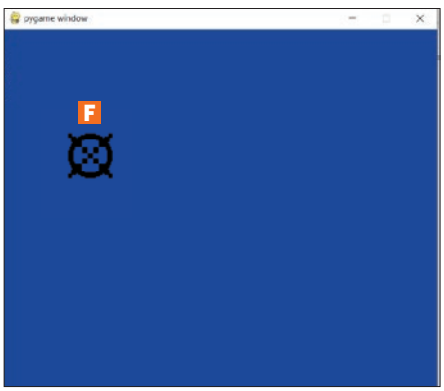
4 Zapisane do tej pory polecenia to jeszcze nie wszystko, co trzeba zrobić, by celownik pojawił się w oknie gry. W warstwie logicznej on już się tam znajduje, jednak by użytkownik zobaczył go w oknie, należy użyć polecenia **pygame.display.update()** D na końcu pętli głównej gry. Sprawi ono, że zawartość okna zostanie odświeżona, czyli wszystkie wprowadzone zmiany w oknie staną się widoczne, a taką zmianą jest właśnie dodanie celownika.

```
okno.blit(celownik, (100, 150))
pygame.display.update()
```
D

5 Zwróćmy jednak uwagę na to, że okno początkowo jest wypełnione czarnym kolorem. Jeśli celownik również jest czarny – jak w przykładzie – nie zauważymy go, choć będzie już tam umieszczony. Dlatego przed instrukcją wstawiającą celownik w oknie dodajmy instrukcję wypełniającą okno kolorem. Będzie ona przydatna, nawet jeśli chcemy mieć okno w kolorze czarnym i celownik w innym kolorze. To dlatego, że wypełnienie okna kolorem z każdym przejściem pętli będzie usuwało grafiki umieszczone w oknie w poprzednich przejściach pętli – to pozwoli na zaprogramowanie ruchu obiektów. Dodajemy polecenie: **okno.fill((0,60,150))** E, w którym (0,60,150) to kolor wyrażony w modelu RGB, gdzie powstaje on z połączenia barwy czerwonej, zielonej i niebieskiej. Kolejne liczby w nawiasie odpowiadają tym właśnie kolorom i mogą przyjmować wartości od 0 do 255. Gdyby każda z nich miała wartość 0, powstałby kolor czarny, natomiast gdyby wszystkie miały wartość 255 – kolor biały. Możemy dowolnie w podanym zakresie łączyć wartości barw, by uzyskiwać różne kolory wypełnienia.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    okno.fill((0,60,150))
    okno.blit(celownik, (100, 150))
```
E

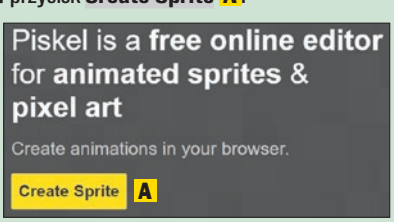
6 Gdy teraz uruchomimy działanie gry, zobaczymy, że celownik jest już widoczny w oknie F.



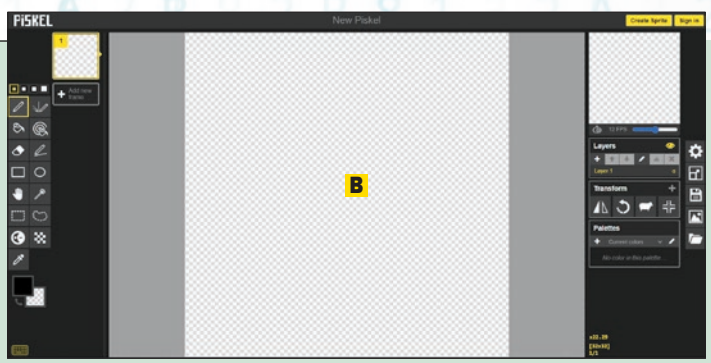
Piskel – edytor graficzny online

Jeśli szukamy programu do tworzenia grafik na potrzeby gier, możemy wykorzystać aplikację **Piskel** – dostępną z poziomu strony **piskelapp.com**.

1 By rozpocząć tworzenie grafiki, wybieramy przycisk **Create Sprite** A.

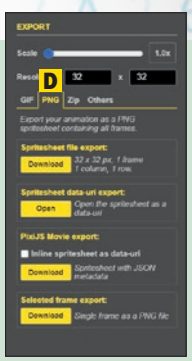


2 Dalej zobaczymy już okno edytora B, w którym narzędziami znajdującymi się w panelu po lewej stronie okna możemy narysować grafikę celownika.



3 Gdy rysowanie jest skończone, należy pobrać ze strony przygotowaną grafikę. W tym celu z panelu po prawej stronie okna wybieramy opcję **Export** C.

4 Na rozwiniętym panelu przechodzimy do zakładki **PNG** D oznaczającej format pliku, w jakim chcemy pobrać utworzoną grafikę.



5 Następnie wybieramy pierwszy przycisk **Download** – plik zostanie pobrany. By móc korzystać z takiego pliku w grze, należy umieścić go w tej samej lokalizacji, w której zapisaliśmy plik ze skryptem naszej gry. Umożliwi nam to odniesienie się do pliku z poziomu skryptu.

RUCH CELOWNIKA W OKNIE

Kolejnym etapem tworzenia naszej gry będzie wprawienie celownika w ruch.

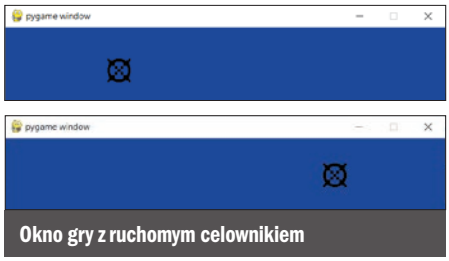
1 Jeśli celownik ma się poruszać w obrębie okna wraz z ruchem kursora myszy, powinniśmy móc odczytywać lokalizację kursora. Zwraca ją polecenie **pygame.mouse.get_pos()**. Gdybyśmy chcieli umieścić celownik w pozycji zwracanej przez to polecenie, to w lokalizacji kursora znalazłby się lewy górny róg grafiki celownika. By lepiej dopasować położenie celownika do kursora, zanim pozycję zwróconą przez polecenie przypiszemy do lokalizacji kursora, nieco ją zmienimy. Przesuniemy grafikę tak, by środek celownika odpowiadał pozycji kursora. Dlatego pozycję kursora zapisujemy do zmiennej, pisząc **poz = pygame.mouse.get_pos()** i umieszczając wspomniane polecenie wewnątrz pętli głównej gry – przed poleceniem umiejscawiającym grafikę celownika w oknie.

```
celownik = pygame.image.load('celownik.png')
pygame.mouse.set_visible(False) B
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

2 Pozycja zapisana pod nazwą **poz** składa się z dwóch liczb – współrzędnej x i y. Dostęp do pierwszej z liczb mamy pod nazwą **poz[0]**, a do drugiej pod nazwą **poz[1]**. Grafika użyta w przykładzie celownika ma 32 piksele szerokości i 32 piksele wysokości. Jeśli celownik umieścimy w pozycji kursora pomniejszonej o połowę szerokości kursora na osi X i połowę wysokości kursora na osi Y, środek celownika będzie ustawiany w lokalizacji kursora. Dlatego pozycję kursora – **(100, 150)** zamieńmy na **(poz[0]-16, poz[1]-16)** A.

```
okno.fill((0,60,150))
poz = pygame.mouse.get_pos()                    A
okno.blit(celownik, (poz[0]-16, poz[1]-16))
pygame.display.update()
```

3 Dodatkowo – kursor w obrębie okna gry nie powinien być widoczny, by nie przysłaniać grafiki celownika. By ukryć kursor w oknie, dodajemy polecenie **pygame.mouse.set_visible(False)** B. Wystarczy, że kursor zostanie ukryty jednokrotnie – nie musimy ukrywać go z każdym odświeżeniem okna – więc polecenie może znaleźć się w kodzie przed pętlą główną gry. Na tym etapie celownik powinien już przemieszczać się po oknie gry.

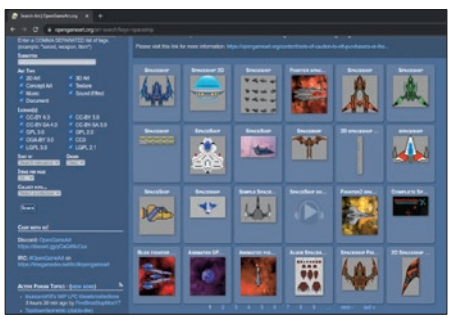


Okno gry z ruchomym celownikiem

WSTAWIENIE STATKÓW KOSMICZNYCH

Jeśli mamy już stworzony celownik, którym możemy sterować, pora na dodanie statków kosmicznych, które powinniśmy zestrzelić.

1 Możemy od razu przygotować grafikę statku kosmicznego. Niech obraz przedstawia statek kosmiczny skierowany ku dołowi okna gry, ponieważ w tym kierunku będzie on lecieć. Jeśli nie chcemy grafiki tworzyć samodzielnie, możemy poszukać obrazu na jednym z portali z wolnymi zasobami – na przykład na stronie **opengameart.org**.



2 Przygotujmy jedną grafikę statku – wykorzystamy ją do umieszczenia w oknie gry kilku statków. Pomoże nam w tym podejście obiektowe.
Warto pamiętać o tym, że Python jest językiem zorientowanym obiektowo, i wykorzystywać możliwość pisania klas i generowania ich obiektów.
Programowanie obiektowe to sposób programowania, w którym programy definiuje się za pomocą obiektów, czyli elementów łączących **stan** (to znaczy dane nazywane **polami**) i **zachowanie** (czyli **procedury** i **funkcje**, w programowaniu obiektowym nazywane **metodami**). Obiektowy program komputerowy rozumiany jest jako zbiór takich obiektów, komunikujących się między sobą w celu wykonywania zadań.
Obiekt jest instancją klasy, a to oznacza, że jeśli chcemy mieć obiekty reprezentujące poszczególne statki kosmiczne, musimy mieć klasę, która definiuje takie statki. Ideę programowania obiektowego najlepiej jest zrozumieć, analizując przykłady. Być może pomoże nam w tym przeanalizowanie klasy napisanej na potrzeby tej gry.

3 Na potrzeby naszej gry przed pętlą główną stworzymy klasę **Statek**, której definicję zaczynamy od **class Statek():** A.

```
pygame.mouse.set_visible
class Statek(): A
while True:
```

4 Podstawą w każdej klasie jest konstruktor – to taka metoda, którą tworzymy nowe obiekty klasy. W Pythonie konstruktor ma nazwę **__init__** – zatem zapisujemy **def __init__():** B. Pamiętajmy przy tym oczywiście o zachowaniu wcięć – metody klasy powinny zawierać wcięcie w kodzie względem definicji klasy.

```
class Statek():
    def __init__(): B
```

5 Każdy obiekt jest widziany sam przez siebie pod nazwą **self**. Nazwy tej używa się wewnątrz klasy w tych miejscach, w których potrzebne jest odwołanie obiektu do niego samego. Nazwa **self** powinna być pierwszym argumentem wszystkich deklarowanych w klasie metod, w tym konstruktora. Musimy dopisać **self** C jako pierwszy parametr konstruktora. W wywołanej metodzie reprezentuje ona aktualnie użyty obiekt.

```
class Statek():
    def __init__(self): C
```

6 Jeśli wywołując konstruktor, mamy tworzyć nowe statki kosmiczne, powinniśmy mieć wpływ na to, co odróżnia poszczególne statki kosmiczne. Takim elementem może być położenie statku w oknie gry. W tym celu musimy podać parametry do konstruktora, tak abyśmy te rzeczy mogli ustawiać, tworząc obiekt, czyli wywołując konstruktor, piszemy zatem kolejne parametry oprócz **self**. Tym razem będzie to **x, y** D, co oznacza, że pierwszym argumentem podawanym przy tworzeniu obiektu będzie współrzędna x, na której ma pojawić się statek, a drugim współrzędna y tego statku.

```
def __init__(self, x, y): D
```

7 Wśród argumentów użytych podczas wywołania **self** nie ma odpowiednika. To oznacza, że nawet po wprowadzeniu tej zmiany pierwszym argumentem podawanym przy tworzeniu obiektu będzie współrzędna x, na której ma pojawić się przeszkoda, a drugim szerokość tej przeszkody.

Wyrwij, użyj dziurkacza i zbieraj do segregatora

8 W treści konstruktora powinniśmy zrobić coś z danymi, które zostaną podane podczas jego uruchamiania. Obiekt klasy ma coś takiego jak pola – są to jego właściwości. Dla naszych statków kosmicznych taką właściwością powinny być współrzędne x i y, na których umieścimy statek kosmiczny. Aby tworzyć właściwości obiektu, w konstruktorze używamy słowa **self** i po kropce piszemy nazwę właściwości. Do tak utworzonego pola klasy powinniśmy przypisać wartość – jeśli tworzymy pole **self.x**, to możemy przypisać do niego wartość bezpośrednio z parametru, poprzez **self.x = x** E.

```
class Statek():
    def __init__(self, x, y):
        self.x = x E
```

9 Podobnie powinniśmy zrobić ze współrzędną y, pisząc **self.y = y** F.

```
def __init__(self, x, y):
    self.x = x
    self.y = y F
```

10 Kolejną właściwością statku kosmicznego powinien być plik graficzny wykorzystany do narysowania statku w oknie gry. W konstruktorze dopisujemy **self.grafika = pygame.image.load('statek.png')** G, gdzie **'statek.png'** to nazwa pliku graficznego ze statkiem kosmicznym, który powinien znajdować się w lokalizacji skryptu gry.

```
self.x = x
self.y = y
self.grafika = pygame.image.load('statek.png') G
```

11 By na dalszym etapie budowania gry móc wygodnie sprawdzać, czy celownik podczas „strzału” nachodził na obszar statku, powinniśmy zdefiniować w konstruktorze obszar zajmowany przez statek – zapisujemy zatem **self.obszar = pygame.Rect(self.x, self.y, 112, 75)** H, gdzie 112 to szerokość, a 75 – wysokość pliku graficznego ze statkiem kosmicznym. Posiadając plik o innych wymiarach, zastąpmy te liczby wymiarami używanej w grze grafiki.

```
self.y = y
self.grafika = pygame.image.load('statek.png') H
self.obszar = pygame.Rect(self.x, self.y, 112, 75)
```

12 Ostatnią właściwością definiującą obiekt musi być informacja, czy statek ma być dalej widoczny w oknie, czy powinien zniknąć. Statek powinien zniknąć, gdy gracz w niego trafi, ale też gdy wyleci z okna gry, by mógł zastąpić go kolejny statek. Właściwość ta może nazywać się **self.czyZnika** i mieć wartość początkową **False** I. Będzie to oznaczało, że statek nowo utworzony nie ma jeszcze znikać. Dopiero gdy pojawi się tam wartość **True**, statek powinien zniknąć z okna gry.

```
self.grafika = pygame.ima
self.obszar = pygame.Rect
self.czyZnika = False I
```

13 Konstruktor jest już gotowy, jednak to nie koniec definicji klasy. Utworzone w przyszłości obiekty, czyli statki kosmiczne, powinniśmy móc umieścić w oknie. W tym celu należy użyć polecenia do wstawiania grafik – jak w przypadku celownika. Jednak tu polecenia tego użyjemy w metodzie **rysujStatek** J, którą musimy utworzyć. Definiując ją, musimy pamiętać o odniesieniu się obiektu do samego siebie, czyli o słówku **self** w nawiasie. Jest ono niezbędne w każdej metodzie klasy.

```
        self.obszar = pygame.Re
        self.czyZnika = False
    def rysujStatek(self): J
```

14 W oknie powinny być narysowane tylko te statki, które jeszcze nie znikają – zatem te, które w polu **self.czyZnika** mają wartość **False**. Można to sprawdzić poprzez instrukcję warunkową **if self.czyZnika == False** K.

```
def rysujStatek(self):                K
    if self.czyZnika == False:
```

15 Gdy warunek z if jest spełniony, można użyć polecenia wstawiającego grafikę statku w oknie gry, w pozycji opisanej poprzez współrzędne x i y statku – zrobimy to, pisząc: **okno.blit(self.grafika, (self.x, self.y))** L.

```
    if self.czyZnika == False:
        okno.blit(self.grafika, (self.x, self.y)) L
```

16 Dysponując tak zdefiniowaną klasą, możemy przejść do tworzenia jej obiektów, by móc w oknie gry zobaczyć statki. Obiekty reprezentujące statki zapiszemy na liście. Listę tworzymy po definicji klasy, pisząc **statki = []** M.

```
        self.obszar
        self.czyZni
    def rysujStatek
        if self.czy
            okno.bl
statki = [] M
```

17 Do listy możemy dodawać obiekty. Pierwszy z nich umieszczamy, pisząc na liście **statki.append(Statek(random.randint(0,szer), 0))** N, gdzie **random.randint(0,szer)** generuje liczbę odpowiadającą odległości nowego statku od lewej krawędzi okna – a będzie to taka liczba, że statek pojawi się w bliżej nieokreślonym miejscu na tej szerokości. Z kolei następne zero to odległość statku od górnej krawędzi okna. Należy jednak pamiętać, że polecenie **random.randint** pochodzi z modułu **random**, który należy importować O do projektu.

```
File  Edit  Format  Run  Options
import pygame
import random O

pygame.init()
szer = 600
```

```
statki = []
statki.append(Statek(random.randint(0,szer), 0))N
```

18 Następne statki możemy importować kolejnymi poleceniami, zmieniając jedynie położenie statku w oknie gry – na przykład: **statki.append(Statek(random.randint(0,szer), 150))** i **statki.append(Statek(random.randint(0,szer), 300))** P.

```
statki = []
statki.append(Statek(random.randint(0,szer), 0))
statki.append(Statek(random.randint(0,szer), 150))
statki.append(Statek(random.randint(0,szer), 300)) P
while True:
```

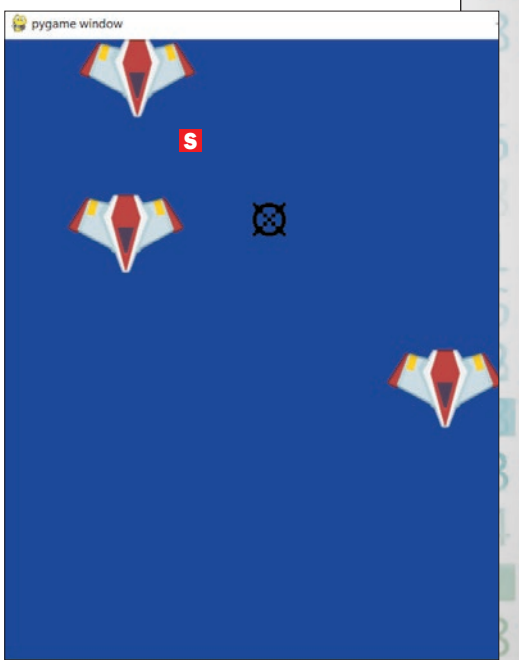
19 By statki stały się widoczne w oknie, należy wywołać metodę **rysujStatek** dla każdego statku dodanego do listy statków. By odnieść się do każdego statku na liście, w pętli głównej dodajemy pętlę **for s in statki:** Q przechodzącą po elementach listy, gdzie s będzie nazwą, poprzez którą odniesiemy się do poszczególnych statków w kolejnych przejściach pętli.

```
okno.blit(celownik, (poz[0]-16
for s in statki: Q
    |
pygame.display.update()
```

20 Żeby wywołać rysowanie, w pętli zapisujemy **s.rysujStatek()** R.

```
okno.blit(celownik, (po
for s in statki:
    s.rysujStatek() R
pygame.display.update()
```

21 Po wprowadzeniu zmian statki są już widoczne S w oknie gry.



RUCH STATKÓW

Do wprawienia statków kosmicznych w ruch będziemy potrzebować nowej metody w klasie **Statek**.

1 Możemy nadać jej nazwę **ruchStatku** A, a w treści dodać polecenie zwiększające wartość współrzędnej y statku o 1.

```
def ruchStatku(self): A
    self.y = self.y + 1
```

2 Jeśli statek pozostanie nietrafiony i wyleci z okna gry, powinniśmy przywrócić go tak, by jeszcze raz przeleciał przez okno, czyli przenieść cały statek ponad okno. Zatem jeśli współrzędna y statku jest większa niż wysokość okna (co sprawdzamy instrukcją warunkową B) – zmieniamy y statku na -75 C.

```
self.y = self.y + 1
if self.y > wys:    B
    self.y = -75    C
```

3 Na koniec należy ponownie zdefiniować obszar zajmowany przez statek, pisząc **self.obszar = pygame.Rect(self.x, self.y, 112, 75)** D, ponieważ po zmianie położenia statku obszar, w którym będziemy wykrywać trafienie **s statek**, również powinien się przemieścić.

```
if self.y > wys:
    self.y = -75
self.obszar = pygame.Rect(self.x, self.y, 112, 75)  D
```

4 By wywołać ruch statków, w pętli głównej do pętli przechodzącej po liście statków dopisujemy **s.ruchStatku()** E.

```
for s in statki:
    s.ruchStatku() E
    s.rysujStatek()
```

5 Pętla główna gry wykonuje się tak często, jak pozwalają na to moce przerobowe komputera, na którym uruchomiono grę. Im mocniejszy jest sprzęt, tym częściej wykonuje się pętla, a co za tym idzie – tym szybsze są statki. By uniezależnić działanie gry od sprzętu, na jakim jest uruchomiona, w pętli głównej gry, przed odświeżaniem zawartości okna, dodajemy polecenie **pygame.time.wait(10)** F. Sprawi ono, że pętla będzie wykonywać się co 10 milisekund.

```
    s.ruchStatku()
    s.rysujStatek()
pygame.time.wait(10) F
```

ZBIJANIE STATKÓW

Statki powinny znikać, gdy w nie strzelimy. W tym celu powinniśmy wychwycić wystąpienie zdarzenia kliknięcia myszą. Sam mechanizm przechodzenia po liście zdarzeń mamy już zaprogramowany. Został on napisany na potrzeby dodania obsługi przycisku zamykającego okno gry. Mechanizm ten należy jednak rozbudować o wykrywanie kolejnego zdarzenia.

1 Wewnątrz pętli przechodzącej po liście zdarzeń dodajemy kolejną instrukcję warunkową **if event.type == pygame.MOUSEBUTTONDOWN:** A.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        quit()
A   if event.type == pygame.MOUSEBUTTONDOWN:
```

```
if event.type == pygame.MOUSEBUTTONDOWN:  B
    pozKlik = pygame.mouse.get_pos()
    for s in statki:
        if s.obszar.collidepoint(pozKlik) and s.czyZnika == False:
            s.czyZnika = True
            statki.append(Statek(random.randint(0,szer), -150))
```

2 Gdy wykryjemy zdarzenie, należy wykonać następujący zestaw instrukcji B, gdzie:

- **pozKlik = pygame.mouse.get_pos()** – to zapisanie pod nazwą **pozKlik** pozycji środka celownika,
- **for s in statki:** – pętla pozwalająca na przejście po liście statków,
- **if s.obszar.collidepoint(pozKlik) and s.czyZnika == False:** – sprawdzenie, czy środek celownika nachodzi na obszar statku – ale tylko dla statków, które nie zniknęły,
- **s.czyZnika = True** – ukrycie statku,
- **statki.append(Statek(random.randint(0,szer), -150))** – wytworzenie nowego statku na liście.

PUNKTY

Możemy już zbijać statki. Każde zbicie statku może dodawać nam punkty.

1 Jeśli chcemy je zliczać, należy utworzyć zmienną, która będzie magazynować liczbę punktów. Tworzymy zmienną **punkty** A o wartości początkowej równej 0 – przed pętlą główną gry.

```
statki.append
punkty = 0 A
while True:
```

2 Wartość zmiennej powinna rosnąć o 1 – przy każdym trafieniu statku B.

3 Zmienna ta może być wykorzystana do określenia wyniku w grze. Załóżmy, że zadaniem gracza będzie zbicie 30 statków w jak najkrótszym czasie. Czas od uruchomienia gry możemy sprawdzić poleceniem **pygame.time.get_ticks()** – zwraca ono liczbę milisekund od startu. Jeśli w pętli głównej gry sprawdzimy, jaka jest wartość czasu, gdy punkty będą większe lub równe 30, to poznamy nasz wynik gry. Im czas ten jest mniejszy, tym nasz wynik jest lepszy. Jeśli warunek w instrukcji warunkowej sformułujemy jako punkty **>= 30**, to będzie on spełniony cały czas po uzyskaniu 30 punktów, a co za tym idzie – czas, w jakim to zrobimy, będzie co chwila zmieniany. Dodajmy przed pętlą główną gry, oprócz zmiennej **punkty**, zmienną o nazwie **czytrwagra** i wartości **True** C.

```
punkty = 0
czytrwagra = True C
while True:
    for event in
```

```
if s.obszar.collidepoint(pozKlik) and s.czyZnika == False:
    punkty = punkty + 1 B
    s.czyZnika = True
```

```
    s.rysujStatek()
if czytrwagra == True and punkty >= 30:  D
    czytrwagra=False
    print("Twój wynik to: "+str(pygame.time.get_ticks()))
pygame.time.wait(10)
```

4 Gdy zmienna ta istnieje, umieszczamy w pętli głównej następującą instrukcję warunkową z zawartymi w niej poleceniami D, gdzie:

- **if czytrwagra == True and punkty >= 30:** – sprawdza, czy gra trwa i czy mamy odpowiednią liczbę punktów, by wygrać,
- **czytrwagra=False** – oznaczamy w skrypcie, że gra została już wygrana,
- **print("Twój wynik to: "+str(pygame.time.get_ticks()))** – wypisuje w oknie konsoli wynik rozgrywki w postaci czasu (w milisekundach), jakiego gracz potrzebował na zestrzelenie 30 statków.

Wyrwij, użyj dziurkacza i zbieraj do segregatora

PYTHON JAKO JĘZYK WYKORZYSTYWANY W EDUKACJI

POPRZEDNIE ODCINKI ZNAJDZIESZ W KŚ+ JAKO E-WYDANIE

Na lekcjach informatyki uczniowie poznają zazwyczaj jeden z dwóch języków programowania – C++ lub Python. Jak wyglądają szkolne zadania dotyczące Pythona? Jest on wykorzystywany do zapisywania wymaganych programem nauczania algorytmów – jak algorytmy sortowania czy szyfrowania danych. W tym odcinku kursu zrealizujemy kilka zadań wzorowanych na szkolnych poleceniach. Taki kurs może być dobrą powtórką – na przykład dla maturzystów.

OBSŁUGA PLIKÓW TEKSTOWYCH

Wiele zadań – szczególnie zadań maturalnych – wymaga wczytania zestawu danych, na których będzie pracował program z pliku tekstowego. Obsługa plików tekstowych w języku Python jest prosta i wygodna. Zobaczmy, jak przebiega ten proces.

ODCZYT PLIKU

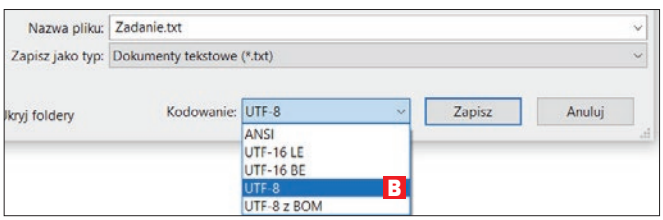
1 Przyjrzyjmy się poleceniu otwierającemu plik – **f = open("Zadanie 2.txt","r")** A. Pod nazwą f w naszym programie zapisany jest odnośnik do danych zawartych w pliku tekstowym o nazwie **Zadanie 2.txt**. Zgodnie z takim zapisem, plik ten, by został otwarty, powinien znajdować się w lokalizacji, w której zapisany jest również sam skrypt. Jeśli plik znajduje się w innej lokalizacji – w miejscu nazwy należałoby umieścić całą ścieżkę do owego pliku.

A

```
File Edit Format Run Options Window Help
f = open("Zadanie 2.txt","r")
```

2 To samo polecenie zawiera jeszcze jeden parametr – w przykładzie była to litera **r**. Jest to skrót od słowa **read** oznaczającego otwarcie pliku w trybie odczytu.

3 Czasem zdarza się, że podczas obsługi plików tekstowych mogą wystąpić problemy z kodowaniem plików. Szczególnie uciążliwe jest to przy obsłudze polskojęzycznych tekstów. Plik zapisany z nieprawidłowym kodowaniem może sprawić, że zamiast polskich znaków widać będzie niezrozumiałe dla nas symbole. Aby naprawić ten problem, wystarczy otworzyć plik z parametrem określającym jego kodowanie. Samo kodowanie pliku tekstowego możemy wybrać podczas jego zapisu w Notatniku B. Mając plik z kodowaniem UTF-8, podczas jego otwierania w Pythonie dodajemy do polecenia otwierającego **encoding="utf-8"** C.



4 By móc wygodnie odnosić się do danych zawartych w pliku tekstowym, możemy skorzystać z pętli **for** zapisanej w formie **for i in f:** – gdzie z każdym przejściem pętli zmienna **i** będzie przyjmowała wartość kolejnej linii tekstu z pliku.

5 Spójrz na przykładowy skrypt D. Tworzy on tablicę o nazwie **T** i każdą linię tekstu z pliku tekstowego dodaje jako nowy element tablicy. W ten sposób tworzymy zbiór danych, na których można realizować algorytmy określone treściami szkolnych zadań, do jakich zbiory danych otrzymaliśmy w plikach tekstowych.

C

```
File Edit Format Run Options Window Help
f = open("Zadanie 2.txt","r",encoding="utf-8")
```

D

```
File Edit Format Run Options Window Help
f = open("Zadanie 2.txt","r",encoding="utf-8")
T = []
for i in f:
    T.append(i)
```

Read i readline

Dla pliku wczytanego jako **f** możemy odczytać jego zawartość również poprzez **f.read()** – użycie takiego polecenia zwróci nam cały tekst z pliku. Jeśli do tego polecenia dołożymy parametr – na przykład **f.read(10)** – zwrócone zostanie pierwsze 10 znaków z pliku. Co ciekawe, kolejne wywołanie polecenia **f.read(10)** zwróci już nie te same 10 znaków, ale kolejne 10 znaków z pliku tekstowego. Podobnie będzie z metodą **readline** – każde kolejne wywołanie **f.readline()** będzie zwracało kolejną linię tekstu z pliku wejściowego.

Fot. Nobi Prizue/iStockphoto.com

ZAPIS DO PLIKU

Wiele zadań wymaga także zapisania wyników działania programu do pliku tekstowego. W celu zapisania danych również otwieramy plik metodą **open**, jednak inny będzie tryb jego otwarcia.

1 Literę **r** z pierwotnego polecenia należy zastąpić literą **w** A, czyli skrótem od słowa **write**. Pamiętajmy jednak o tym, że jeśli chcemy pracować na kilku plikach, nazwy **f** używamy tylko do jednego z nich, a drugiemu należy nadać już inną nazwę i kontynuować przyjęte nazewnictwo w dalszej części skryptu. Co dość ważne, plik o podanej przez nas nazwie nie musi istnieć – Python może sam wytworzyć plik na potrzeby zapisania w nim danych.

A
```
File Edit Format Run Options Window Help
f = open("wyniki.txt","w")
```

2 W przypadku zapisywania danych do pliku możemy skorzystać z polecenia **f.write("Pierwsze dane")** – gdzie podajemy, co chcemy wpisać do pliku. W przypadku użycia pliku, który zawiera już dane, zostaną one zastąpione podanymi przez nas nowymi danymi.

3 Każde kolejne użycie **f.write()** w tym samym programie będzie dopisywało dane do pliku. Co ważne, dane dopisywane są ciągiem. To oznacza, że jeśli użyjemy w kolejnych liniach poleceń **f.write("Pierwsze dane")** i **f.write("Drugie dane")** B, w pliku pojawi się tekst **Pierwsze daneDrugie dane**.

B
```
File Edit Format Run Options Window Help
f = open("wyniki.txt","w")
f.write("Pierwsze dane")
f.write("Drugie dane")
```

4 Nie zdziwmy się jednak, jeśli po próbie uruchomienia przykładowego skryptu program nie zapisze danych do pliku. By dane pojawiły się ostatecznie w pliku, program powinien skończyć pracę na danych, czyli zamknąć otwarty plik. W tym celu dopisujemy **f.close()** C.

C
```
File Edit Format Run Options Window Help
f = open("wyniki.txt","w")
f.write("Pierwsze dane")
f.write("Drugie dane")
f.close()
```

5 Po uruchomieniu takiego programu w lokalizacji, w jakiej zapisano skrypt, pojawi się już plik **wyniki.txt**.

6 W jego zawartości zobaczymy tekst **Pierwsze daneDrugie dane** D.

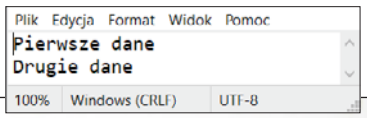
D

7 By dane zapisywane były w oddzielnych liniach, skorzystajmy ze sposobu, jakim jest wprowadzenie do tekstu znaku nowej linii, który w formie tekstowej zapisujemy jako **\n**

8 Jeśli wprowadzimy go do wcześniej użytego polecenia **f.write("Pierwsze dane\n")** E, na końcu linii (w miejscu jego wprowadzenia) otrzymamy znak nowej linii.

E
```
File Edit Format Run Options Window Help
f = open("wyniki.txt","w")
f.write("Pierwsze dane\n")
f.write("Drugie dane")
f.close()
```

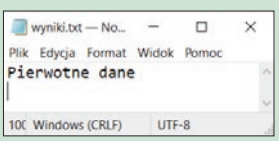
9 Po otwarciu pliku tekstowego **wyniki.txt** zobaczymy, że dane są już zapisane w oddzielnych liniach.



Tryb otwarcia „a”

Pliki możemy otwierać także w trybie **a** – plik zostanie otwarty do zapisu, jednak nie zastępuje istniejącej w nim już treści, ale dodaje nową treść na końcu pliku.

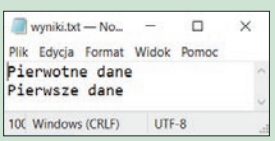
1 Zakładamy, że mamy plik **wyniki.txt**, który zawiera już linię tekstu.



2 Otwieramy ten plik w trybie **append** i wpisujemy do pliku linię danych **Pierwsze dane**.

```
File Edit Format Run Options Window Help
f = open("wyniki.txt","a")
f.write("Pierwsze dane")
f.close()
```

3 Po uruchomieniu skryptu dane zostaną dopisane do pierwotnej treści pliku **wyniki.txt**.



REKURENCJA

Jednym z poruszanych w szkołach programistycznych zagadnień jest rekurencja, zwana też rekursją. Polega ona na odwoływaniu się funkcji czy też definicji do samej siebie. Mechanizm ten jest wykorzystywany w niektórych algorytmach – jak obliczanie wartości silni czy też wyrazu ciągu Fibonacciego. Poznajmy najważniejsze zasady korzystania z tych możliwości.

SILNIA

Silnia liczby naturalnej **n** – to iloczyn kolejnych liczb naturalnych od 1 do n. Silnię liczby naturalnej w matematyce n oznaczamy symbolem **n!**. Przykładowo – silnię liczby 5 zapiszemy jako 5!, co równe jest wynikowi działania 1*2*3*4*5, czyli 120. Natomiast 6! moglibyśmy zapisać jako wynik działania 1*2*3*4*5*6. Jak łatwo zauważyć, znając silnię liczby 5, możemy pomnożyć ją razy 6 i otrzymać w ten sposób wartość silni liczby 6, co zapiszemy działaniem 6! = 5!*6. To oznacza, że aby obliczyć silnię podanej liczby, wystarczy podaną liczbę pomnożyć razy silnię liczby o jeden mniejszej. Tę właśnie zależność można wykorzystać podczas budowy rekurencyjnej funkcji.

1 Definiujemy funkcję o nazwie **silnia** przyjmującą jako parametr liczbę **n** A.

A B
```
File Edit Format Run Options Window Help
def silnia(n):
    return n*silnia(n-1)
```

2 Funkcja powinna zwracać (return) wartość wywołania funkcji **silnia** dla liczby o jeden mniejszej od n, pomnożonej razy n B.

3 W przypadku funkcji rekurencyjnych musimy w pewnym momencie zatrzymać kolejne rekurencyjne wywołania, by funkcja nie wywoływała sama siebie w nieskończoność. Kiedy należałoby zatrzymać rekurencję? Do obliczenia silni liczby 1 nie potrzebujemy obliczania silni dla 0. Nasza funkcja dla liczby **n** równej 1 może od razu zwrócić wartość 1. Zapiszemy to na początku funkcji, korzysta-

Wyrwij, użyj dziurkacza i zbieraj do segregatora

jąc z instrukcji warunkowej – sprawdzającej, czy n jest równe 1 C i gdy tak się stanie, zwracającej wartość 1 poprzez polecenie **return 1** D.

```
def silnia(n):
    if n == 1:C
        return 1 D
    return n*silnia(n-1)
```

4 Uruchamiamy program zawierający definicję funkcji, a następnie w oknie konsoli wpisujemy **silnia(5)** E. W odpowiedzi zobaczymy wartość **120**, czyli prawidłowo obliczoną wartość silni liczby 5.

```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26
:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" fo
r more information.
>>>
===== RESTART: C:/Users/konra/AppData/Local/Programs/
Python/Python39/ks.py =====
>>> silnia(5) E
120
```

CIĄG FIBONACCIEGO

Innym zadaniem, które możemy rozwiązać w sposób rekurencyjny, jest obliczanie wyrazów ciągu Fibonacciego. Jest to ciąg liczb naturalnych określony w taki sposób, że pierwszy wyraz ciągu jest równy 0, drugi jest równy 1, a każdy następny jest sumą dwóch poprzednich. Zatem początkowe wyrazy tego ciągu to: 0 1 1 2 3 5 8 13 21...

1 Definiujemy funkcję **fib** A, która będzie przyjmowała jeden parametr – liczbę n, mówiącą o tym, który element ciągu chcemy obliczyć. Wiemy już, jakie wartości mają zerowy i pierwszy element ciągu. Zatem jeśli n jest mniejsze niż 1, powinniśmy zwrócić wartość 0 – co można zrealizować, umieszczając w definicji instrukcję warunkową.

```
A def fib(n):
      if n < 1:
          return 0
```

2 Podobnie jest z kolejnym wyrazem ciągu – jeśli n < 2 B, zwracamy wartość 1.

```
if n < 1:
    return 0
if n < 2:B
    return 1
```

3 W pozostałych sytuacjach wartości ciągu obliczamy już w sposób rekurencyjny, czyli zwracamy sumę wartości zwróconych przez **fib(n - 1)** i **fib(n - 2)** C.

```
def fib(n):
    if n < 1:
        return 0
    if n < 2:
        return 1
    return fib(n - 1) + fib(n - 2) C
```

4 Przetestujmy funkcję, wpisując w konsoli wywołanie **fib(8)** – co powinno zwrócić liczbę **21** D.

```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.192
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more informat
>>>
===== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python39
>>> fib(8)
21 D
>>>
```

SORTOWANIE

W Pythonie mamy gotowe metody, którymi możemy sortować elementy listy. Przykładowo, mając listę liczb x = [6,8,2,5,9], możemy ją posortować, wywołując metodę sort – pisząc **x.sort()** A. Wypisanie poprzez **print(x)** zawartości tej listy sprawi, że w konsoli zobaczymy elementy listy uporządkowane niemalejąco B.

```
x = [6,8,2,5,9]
x.sort() A
print(x)
```

```
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python39/ks.py =====
[2, 5, 6, 8, 9]
>>>
B
```

Od uczniów wymaga się jednak często, by znali algorytmy sortowania i potrafili zapisać je w językach programowania. Przydatne działania znajdziemy w poniższej poradzie.

SORTOWANIE BĄBELKOWE

Jednym z przykładowych algorytmów sortowania jest sortowanie bąbelkowe. Polega ono na porównywaniu parami elementów listy i zamienianiu ich kolejnością, jeśli pierwszy z elementów jest większy od drugiego. Dla potrzeb naszego zadania możemy przyjąć, że listą do posortowania jest lista x, która została już wspomniana we wcześniejszej części tego tekstu. Pierwsza parą, jaką należy porównać, jest 6 i 8. 6 jest mniejsze, więc nie zamieniamy liczb kolejnością. Kolejna para to 8 i 2 – pierwsza z liczb jest większa, więc zamieniamy liczby kolejnością, a lista powinna mieć po tym kroku postać [6,2,8,5,9] – zatem kolejną parę tworzą liczby 8 i 5. Tu kolejny raz dokonujemy zamiany ich miejscami i uzyskujemy listę [6,2,5,8,9] – gdzie kolejną parę tworzą liczby 8 i 9, jednak tu pierwsza z liczb jest mniejsza i zamiany nie dokonujemy. W ten sposób ostatnim elementem listy jest element największy, jednak nie wszystkie elementy są jeszcze uporządkowane. By mieć pewność, że wszystkie elementy są już uporządkowane, powinniśmy zapętlić porównywanie liczb parami od początku. Takich przejść po liście powinniśmy wykonać o jedno mniej niż liczba elementów listy. Ten mechanizm łatwo zapisać w języku programowania.

```
x = [6,8,2,5,9] A

def bubble(lista): B
```

1 Definiujemy listę **x** A, jak we wcześniejszym przykładzie, i funkcję **bubble** B, w której zapiszemy opisany mechanizm. Funkcja powinna przyjmować jeden parametr – przyjmujący listę elementów do uporządkowania.

2 W funkcji umieszczamy pętlę **for** C, która wykona się o jeden raz mniej niż liczba elementów listy – tyle będziemy wykonywać przejść po całej liście.

```
x = [6,8,2,5,9]

def bubble(lista):
    C for i in range(len(lista)-1):
```

```
def bubble(lista):
    for i in range(len(lista)-1):
D       for j in range(len(lista)-1):
```

3 W pętli umieszczamy kolejną pętlę **for** D, która znów wykona się o jeden raz mniej niż liczba elementów listy. Ta pętla odpowiada za tworzenie kolejnych par liczb do porównania – par jest zawsze o jedną mniej niż liczb na liście.

4 W wewnętrznej pętli umieszczamy instrukcję warunkową **if lista[j] > lista[j+1]:** E, która sprawdzi, czy pierwszy element z pary jest większy od kolejnego elementu z pary.

```
File  Edit  Format  Run  Options  Window  Help
x = [6,8,2,5,9]

def bubble(lista):
    for i in range(len(lista)-1):
        for j in range(len(lista)-1):
E           if lista[j] > lista[j+1]:
```

5 Jeśli warunek instrukcji warunkowej jest spełniony, należy zamienić elementy listy miejscami. Python posiada wygodny mechanizm do tego przeznaczony – wystarczy zapisać **lista[j],lista[j+1] = lista[j+1],lista[j]** F.

```
def bubble(lista):
    for i in range(len(lista)-1):
        for j in range(len(lista)-1):
            if lista[j] > lista[j+1]:
F               lista[j],lista[j+1] = lista[j+1],lista[j]
    return lista
G print(bubble(x)) H
```

6 Na końcu funkcji umieszczamy jeszcze polecenie **return lista** G, by funkcja zwróciła w wyniku swojego działania uporządkowaną listę.

7 Przetestujmy działanie programu, po dopisaniu na jego końcu polecenia **print(bubble(x))** H, które powinno sprawić, że w oknie konsoli wypisana zostanie uporządkowana lista I.

```
IDLE Shell 3.9.6
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python39/ks.py =====
[2, 5, 6, 8, 9] I
>>>
```

WYSZUKIWANIE NAJMNIEJSZEGO I NAJWIĘKSZEGO ELEMENTU W ZBIORZE

Podobnie jak z sortowaniem do wyszukiwania minimum i maksimum wśród elementów listy możemy skorzystać odpowiednio z funkcji min i max. Dla listy, której używaliśmy w poprzednim zadaniu, polecenie **print(min(x))** A wypisze w konsoli B najmniejszy, a polecenie **print(max(x))** największy element listy.

Warto jednak znać algorytm opisujący ten mechanizm.

```
File  Edit  Format  Run  Options  Window  Help
x = [6,8,2,5,9]
print(min(x)) A
print(max(x))
```

```
IDLE Shell 3.9.6
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/konra/AppData/Local/Programs/Python/Python39/ks.py =====
2 B
9
>>>
Ln: 7  Col: 4
```

POSZUKIWANIE NAJMNIEJSZEGO ELEMENTU LISTY

1 Na potrzeby zadania wykorzystamy ponownie listę **x**. Do programu, w którym jest ona zdefiniowana, dodajemy definicję funkcji **najmniejszy** przyjmującą jako parametr listę A.

```
File  Edit  Format  Run  Options  Window  Help
x = [6,8,2,5,9]

def najmniejszy(lista): A
```

2 Deklarujemy w funkcji zmienną **y**, która będzie służyła do przechowywania aktualnie najmniejszej liczby, jaką udało się znaleźć. Na początek jej wartością powinien być pierwszy element listy – zapisujemy zatem **y = lista[0]** B.

```
def najmniejszy(lista):
    y = lista[0] B
```

3 Dalej należy porównywać wartość zmiennej **y** z kolejnymi elementami listy. By takie porównanie stworzyć i odnosić się do kolejnych elementów listy, potrzebujemy pętli **for** C, która pozwoli pod nazwą **i** umieszczać w swoich kolejnych przejściach elementy listy.

```
def najmniejszy(lista):
    y = lista[0]
C   for i in lista:
```

4 Wewnątrz pętli można już utworzyć instrukcję warunkową sprawdzającą, czy **i** jest mniejsze niż **y - if i< y:** D.

5 Jeśli warunek jest spełniony, nową wartością **y** powinna być obecna wartość **i** E.

```
    y = lista[0]
    for i in lista:
        if i < y: D
```

```
    for i in lista:
        if i < y:
            y = i E
```

6 Po przejściu pętli zwracamy wartość zmiennej **y** – jako najmniejszej wartości na liście – pisząc **return y** F.

```
def najmniejszy(lista):
    y = lista[0]
    for i in lista:
        if i < y:
            y = i
    return y F
```

7 By przetestować działanie funkcji, dodajemy do programu jej wywołanie, umieszczone w instrukcji **print** G – co sprawi, że wartość ta zostanie wypisana w oknie konsoli H, po uruchomieniu programu.

```
    for i in lista:
        if i < y:
            y = i
    return y

print(najmniejszy(x)) G
```

```
File  Edit  Shell  Debug  Options  Windov
Python 3.9.6 (tags/v3.9.6
D64)] on win32
Type "help", "copyright",
>>>
===== RESTART: C:/Users/k
2 H
>>>
```

POSZUKIWANIE NAJWIĘKSZEGO ELEMENTU LISTY

Opisywany algorytm poszukiwania najmniejszego elementu listy bardzo łatwo przerobić na poszukiwanie największego elementu. Wystarczy, że w instrukcji warunkowej zmienimy sprawdzanie, czy kolejny element listy jest większy I od zapisanego jako **y** elementu. W ten sposób, po wykonaniu przejścia po wszystkich elementach listy, jako **y** zachowamy ten największy.

```
for i in lista:
    if i > y: I
        y = i
return y
```

Wyrwij, użyj dziurkacza i zbieraj do segregatora

BAZY DANYCH W PYTHONIE

POPRZEDNIE ODCINKI ZNAJDZIESZ W KŚ+ JAKO E-WYDANIE

Ogrom dostępnych dla aplikacji informacji, których analiza może przynosić olbrzymie korzyści, sprawia, że niezwykle cenne są możliwości ich magazynowania i przetwarzania. I właśnie do tego służą bazy danych. Bazy danych stanowią ważną część wielu rodzajów dostępnego oprogramowania.

Z tej części kursu dowiemy się, jak pracować z bazami danych z poziomu programów pisanych w języku Python.
Istnieje wiele zarówno systemów do zarządzania bazami danych, jak i typów baz danych. W tym poradniku zajmiemy się **relacyjnymi bazami danych** i platformą **sqlite3**.

Czym są relacyjne bazy danych i jak można je wykorzystywać w projektach, dowiemy się z przedstawionego przykładu.

QUIZ, GRA OPARTA NA BAZIE PYTAŃ

Gry typu quiz polegają na udzielaniu przez użytkownika odpowiedzi na zadawane pytania. Mogą to być na przykład pytania zamknięte. Do każdego pytania musi być wtedy przyporządkowanych kilka odpowiedzi do wyboru, z których tylko jedna jest prawidłowa. W takich grach największym wyzwaniem jest to, w jaki sposób można zmagazynować odpowiednio dużo pytań i od powiedzi. W naszym przykładzie do ich przechowywania wykorzystamy właśnie bazę danych.

PODSTAWY WYKORZYSTANIA MODUŁU SQLITE3

Sqlite3, czyli moduł do obsługi relacyjnych baz danych w Pythonie, z którego będziemy korzystać w tym poradniku, jest modułem wbudowanym, a to oznacza, że nie musimy go dodatkowo instalować.
Co jednak ważne, bazy danych z tego modułu możemy obsługiwać nie tylko z poziomu kodu aplikacji w Pythonie, ale też z poziomu zewnętrznej aplikacji, która wymaga doinstalowania.

1 Wracając jednak do podstaw, by móc korzystać z modułu, należy go importować, co zrobimy, pisząc na początku naszego skryptu **import sqlite3** A.

```
*zbazą.py - D:/Users/konra/Documents/zbazą.py (3.9.6)*
File  Edit  Format  Run  Options  Window  Help
import sqlite3
```
A

2 Gdy moduł został już importowany, należy nawiązać połączenie z plikiem przechowującym naszą bazę danych. Samo połączenie również może mieć swoją nazwę, zazwyczaj używa się dla niego nazwy **conn**. Dlatego od niej rozpoczynamy instrukcję **conn = sqlite3.connect('e.sqlite')** B, gdzie **e.sqlite** to nazwa i rozszerzenie pliku przechowującego naszą bazę danych. Co istotne, plik ten nie musi wcześniej istnieć, po tak zapisanej instrukcji podany przez nas plik zostanie utworzony w lokalizacji, w której przechowujemy nasz program.

```
File  Edit  Format  Run  Options  Window  Help
import sqlite3
B
conn = sqlite3.connect('e.sqlite')
```

3 Kolejna linia, **c = conn.cursor()** C, to tak zwane utworzenie kursora, czyli obiektu pozwalającego na wywoływanie zapytań SQL do naszej bazy.

```
conn = sqlite3.conn

c = conn.cursor()C
```

4 Te z kolei mogą polegać na tworzeniu tabel, wstawianiu do nich danych, usuwaniu ich, aktualizacji czy też po prostu wyciąganiu danych z tabel. Dysponując pustą bazą, należy utworzyć w niej tabelę, ponieważ to w tabelach gromadzone są dane. Na potrzeby gry typu quiz dobrze sprawdzi się tabela, w której każdy wiersz danych będzie zawierał: numer pytania, treść pytania, cztery możliwe do wyboru odpowiedzi i jedną z odpowiedzi zapisaną jako poprawną. Każda z tych danych powinna być przechowywana w oddzielnej kolumnie tabeli. By móc w ogóle utworzyć tabelę, należy dla utworzonego wcześniej kursora wywołać polecenie execute, co zrobimy, pisząc **c.execute()** D.

```
c = conn.cursor()
c.execute()D
```

5 Jako atrybut tego polecenia zapiszemy w języku SQL polecenie tworzące wspomnianą wcześniej tabelę. Tym poleceniem jest **CREATE TABLE pytania** E – zapisujemy je w cudzysłowie. Słowo **pytania** w tym poleceniu oznacza nazwę tworzonej tabeli.

```
c = conn.cursor()
c.execute("CREATE TABLE pytania")E
```

6 Nie jest to jednak jeszcze całe polecenie, ponieważ nie określamy w nim, jakie kolumny ma zawierać tworzona przez nas tabela. To wypisujemy w nawiasie, podając dodatkowo, dla każdej kolumny, typ przechowywanych przez nią danych. I tak dopisujemy w cudzysłowie nawias, a w nim: **numer INTEGER, tresc TEXT, odpA TEXT, odpB TEXT, odpC TEXT, odpD TEXT, poprawna TEXT** F, gdzie **INTEGER** oznacza,

```
c.execute("CREATE TABLE pytania (numer INTEGER, tresc TEXT, odpA TEXT, odpB TEXT, odpC TEXT, odpD TEXT, poprawna TEXT)")F
```

Fot. Nobi Plizuer/iStockphoto.com

że w kolumnie o nazwie **numer** będziemy mogli przechowywać tylko liczby całkowite, a **TEXT** przy nazwach pozostałych kolumn oznacza, że będziemy mogli wpisywać tam dowolne teksty.

7 By zapisane wcześniej polecenie się wykonało, dopisujemy w kolejnej linii skryptu **conn.commit()** G.

```
c = conn.cursor()
c.execute("CRE
conn.commit()G
```

8 Na końcu skryptu dopisujemy też **conn.close()** H, by zamknąć połączenie z bazą danych.

9 Po uruchomieniu tak zapisanego skryptu program powinien wykonać się, nie

```
c.execute("CREATE TAB
conn.commit()
conn.close()H
```

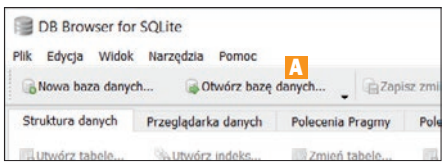
prezentując nam żadnych efektów swojego działania.

W dalszej części poradnika przeczytamy, jak sprawdzić, czy baza i tabela utworzyły się poprawnie.

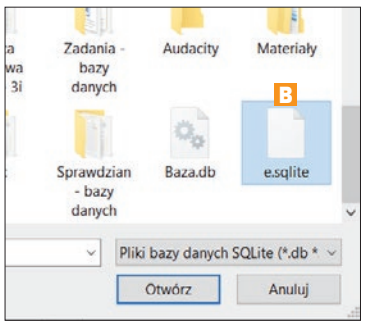
DB BROWSER FOR SQLITE

Najwygodniejszym sposobem na zapoznanie się z zawartością pliku zawierającego bazę danych jest otwarcie go poprzez aplikację **DB Browser for SQLite** (W KŚ+).

1 W programie z menu górnego wybieramy opcję **Otwórz bazę danych** A.

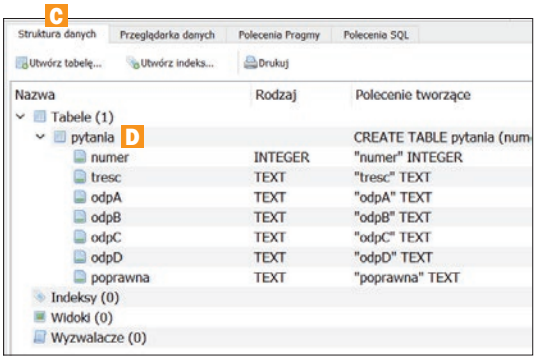


2 Następnie w oknie dialogowym wskazujemy plik **e.sqlite** B znajdujący się w lokalizacji skryptu napisanego przez nas w poprzedniej części poradnika.



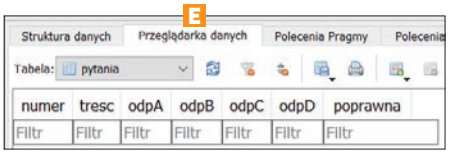
3 Po jego otwarciu w głównej części okna programu, w zakładce **Struktura danych** C, zobaczymy zawartość bazy. W sekcji **Tabele** powinna znajdować się jedna tabela – **pytania** D. Po rozwinięciu jej możemy zobaczyć, z jakich kolumn jest zbudowana.

C



4 Mając tak przygotowaną tabelę, możemy wypełnić ją treścią pytań i odpowiedzi. Warto wiedzieć, że zarówno tworzenie bazy czy tabeli, jak i wypełnianie ich danymi, możemy przeprowadzać w trybie tekstowym, z poziomu skryptu Pythona, ale też w trybie graficznym, z poziomu DB Browser for SQLite. By wypełnić tabelę danymi, przechodzimy w programie do zakładki **Przeglądarka danych** E.

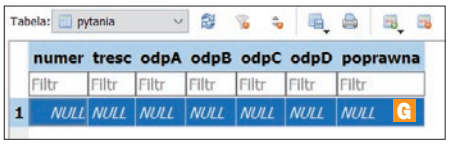
E



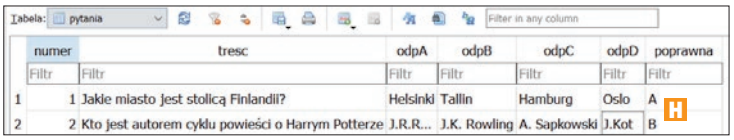
5 By móc wpisać nowy wiersz do tabeli, klikamy na przycisk **Wstaw nowy rekord bieżącej tabeli** F.

F

6 W wyświetlonej tabeli pojawi się linia wypełniona wartościami **NULL** G, w których miejsce wpisujemy właściwe dane. Na tym etapie tworzenia gry typu quiz zastanowimy się, jakie pytania chcemy umieścić w swojej grze. Wpisujemy kolejne wiersze do tabeli, powtarzając czynność wstawiania nowych rekordów do bieżącej tabeli. Zwróćmy uwagę na to, by w ostatniej z kolumn wskazać poprawną odpowiedź na podane obok pytanie, wpisując literę A, B, C lub D H, zależnie od tego, która odpowiedź jest prawidłowa.



7 Po wprowadzeniu odpowiedniej liczby pytań – do stworzenia gry będzie ich potrzeba minimum 10 – kombinacją klawiszy Ctrl+S zapisujemy wprowadzone do bazy zmiany.



WYCIĄGANIE DANYCH Z BAZY

Z tej części poradnika dowiemy się, jak z poziomu skryptu napisanego w Pythonie odnieść się do pytań i odpowiedzi wprowadzonych do bazy poprzez program **DB Browser for SQLite**. Zwróćmy uwagę na to, że program napisany w pierwszej części tego odcinka kursu spełnił już swoje zadanie – baza danych i potrzebna tabela są już utworzone. Dlatego dalsze skrypty tworzące grę zapiszemy w nowym pliku.

1 Skrypt w swojej początkowej formie będzie stosunkowo podobny do tego, jaki napisaliśmy w pierwszej części tego odcinka kursu. To dlatego, że powinien zawierać import modułu **sqlite3**, a następnie połączenie z bazą i utworzenie kursora, dlatego pierwsze trzy linie kodu A możemy powtórzyć.

```
File Edit Format Run Options Window Help
import sqlite3

conn = sqlite3.connect('e.sqlite')

c = conn.cursor()
```
A

2 Kolejna linia kodu powinna polegać na wywołaniu polecenia, które z tabeli **pytania** wyciągnie wszystkie potrzebne nam dane. Polecenie języka SQL, podobnie jak w przypadku tworzenia tabeli, zapiszemy w nawiasie, w cudzysłowie po **c.execute** B.

```
c = conn.cursor()
c.execute("") B
```

3 Tym razem potrzebujemy polecenia wyciągającego dane, czyli **SELECT * from pytania** C. W strukturze tego polecenia * oznacza, że chcemy wyciągnąć dane ze wszystkich kolumn.

```
c = conn.cursor()
c.execute("SELECT * from pytania")C
```

Wyrwij, użyj dziurkacza i zbieraj do segregatora

4 Aby móc te dane wygodnie przeglądać, nie wystarczy, że polecenie jedynie je zwróci, powinniśmy je także zapisać. Dlatego przed **c.execute** powinniśmy podać nazwę i znak równości, by pod podaną nazwą przechować wiersze uzyskane w wyniku wyciągania danych z bazy. W przedstawionym skrypcie nazwą, pod którą przechowamy dane, jest **a** D.

```
c = conn.cursor()
a = c.execute("SELECT * from pytania")
```

5 Żeby sprawdzić, czy dane zostały prawidłowo wyciągnięte z bazy, możemy utworzyć pętlę **for**, która przejdzie po wszystkich elementach **a** E i wypisze poprzez **print** F zawartość każdego wyciągniętego elementu.

```
a = c.execute("

for i in a:
    print(i)
```

6 Dla uproszczenia testowania i poprawienia czytelności w bazie danych przykładowego projektu znajdują się tylko dwa pytania G. Zwróćmy jednak uwagę na to, że cały wiersz wyciągany z bazy zawiera nie tylko treść pytania, ale też treść wszystkich odpowiedzi do wyboru, a także wskazuje nam od razu na poprawną odpowiedź. Sposobem wyświetlania pytań i sprawdzaniem poprawności odpowiedzi udzielonej przez gracza zajmiemy się w dalszej części tego poradnika.

```
for i in a:
    print(i)

conn.close() H
```

7 Kończąc wyciąganie danych z bazy, powinniśmy jeszcze dopisać do naszego skryptu jedną linię zamykającą połączenie z plikiem bazy danych, czyli **conn.close()** H, która powinna znajdować się na końcu naszego skryptu.

```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: D:/Users/konra/Documents/zbazą.py ==========
(1, 'Jakie miasto jest stolicą Finlandii?', 'Helsinki', 'Tallin', 'Hamburg', 'Os
o', 'A')
(2, 'Kto jest autorem cyklu powieści o Harrym Potterze', 'J.R.R. Tolkien', 'J.K.
 Rowling', 'A. Sapkowski', 'J.Kot', 'B')
>>>
```

G

QUIZ W **WERSJI TEKSTOWEJ**

Nasz program potrafi już poprawnie wyciągać dane z bazy. W dalszej części przeczytamy, jak zaprogramować sam mechanizm gry bazujący na wyciągniętych z bazy pytaniach.

1 Pierwszym krokiem powinno być rozbicie wyciągniętego z bazy ciągu znaków zawierającego cały wiersz i dane zapisane w każdej z kolumn na oddzielne części. Tak naprawdę dane tylko z pozoru są połączone. Jeśli będziemy odnosić się do danych poprzez indeksy, będziemy mogli odczytywać tylko dane z kolumny o podanym numerze. Tabela **pytania** ma 7 kolumn, które ponumerowane są od 0. Drugą kolumną, o indeksie 1, jest kolumna zawierająca treść pytania. Jeśli zmodyfikujemy polecenie **print** z pętli **for** tak, by treścią do wypisania zamiast wartości **i** było **i[1]** A, zostaną wypisane tylko dane B z drugiej kolumny dla każdego wiersza.

```
for i in a:
    print(i[1]) A
```

2 Pętla powinna pozwolić także na wypisanie możliwych do wyboru odpowiedzi. Dopisujemy w pętli kolejne polecenie **print**, tym razem wypisujące wartość **"A) " + i[2]**. Sprawi to, że po wypisaniu treści pytania zobaczymy w kolejnej linii literę **A** i treść przypisanej do tej opcji odpowiedzi C.

3 Podobnie robimy z kolejnymi odpowiedziami do wyboru przypisanymi do liter

```
========== RESTART: D:/Users/konra/Documents/zbazą.py ==========
Jakie miasto jest stolicą Finlandii?
Kto jest autorem cyklu powieści o Harrym Potterze B
```

```
========== RESTART: D:/Users/konra/Documents/zbazą.py ==========
Jakie miasto jest stolicą Finlandii?
A) Helsinki
Kto jest autorem cyklu powieści o Harrym Potterze
A) J.R.R. Tolkien C
>>>
```

```
========== RESTART: D:/Users/konra/Documents/zbazą.py ==========
Jakie miasto jest stolicą Finlandii?
A) Helsinki
B) Tallin
C) Hamburg      E
D) Oslo
Kto jest autorem cyklu powieści o Harrym Potterze
A) J.R.R. Tolkien
B) J.K. Rowling
C) A. Sapkowski
D) J.Kot
>>>
```

B, C i D D – wyświetlamy przy tych literach zawartość kolejnych kolumn.

```
for i in a:
    print(i[1])
    print("A) "+i[2])
D   print("B) "+i[3])
    print("C) "+i[4])
    print("D) "+i[5])
```

4 Jak widać, po uruchomieniu programu w konsoli będą wyświetlone już wszystkie odpowiedzi do wyboru. Zwróćmy jednak uwagę, że program od razu wypisuje wszystkie dostępne pytania E, a użytkownik nie ma możliwości udzielenia na nie odpowiedzi.

```
for i in a:
    print(i[1])
    print("A) "+i[2])
    print("B) "+i[3])
    print("C) "+i[4])
    print("D) "+i[5])
F   udzielona = input()
```

5 By można było udzielać odpowiedzi w trybie tekstowym, w kodzie naszego programu powinno się pojawić polecenie **input**. Przed nim podajemy nazwę zmiennej, do której ma trafić podany przez użytkownika tekst. Na przykład **udzielona = input()** F. Po jej wprowadzeniu program wyświetli w pętli pierwsze pytanie i odpowiedzi, a następnie będzie oczekiwał, aż użytkownik wprowadzi odpowiedź na pierwsze z pytań. Warto rozbudować polecenie **input** tak, by podczas oczekiwania na udzielenie odpowiedzi wyświetliło informację dla użytkownika, co tak właściwie ma on zrobić. Tekst dodatkowy do wyświetlenia podajemy w nawiasie, w cudzysłowie po **input**, na przykład **"Wybierz odpowiedź A, B, C lub D: "** G.

```
print("C) "+i[4])
print("D) "+i[5])
udzielona = input("Wybierz odpowiedź A, B, C lub D: ")
```

G

6 Po uruchomieniu program wyświetli treść kolejnego pytania i odpowiedzi na nie dopiero po udzieleniu odpowiedzi na pierwsze z pytań H.

```
================== RESTART: D:/Users/konra/Document
Jakie miasto jest stolicą Finlandii? H
A) Helsinki
B) Tallin
C) Hamburg
D) Oslo
Wybierz odpowiedź A, B, C lub D: A
Kto jest autorem cyklu powieści o Harrym Potterze
A) J.R.R. Tolkien
B) J.K. Rowling
C) A. Sapkowski
D) J.Kot
Wybierz odpowiedź A, B, C lub D:
```

```
================== RESTART: D:/Users/konra/Documents
Jakie miasto jest stolicą Finlandii?
A) Helsinki
B) Tallin
C) Hamburg
D) Oslo
Wybierz odpowiedź A, B, C lub D: A
Kto jest autorem cyklu powieści o Harrym Potterze
A) J.R.R. Tolkien
B) J.K. Rowling
C) A. Sapkowski
D) J.Kot
Wybierz odpowiedź A, B, C lub D: B
Twój wynik to: 2 M
>>>
```

7 W grze powinniśmy w jakiś sposób zdobywać punkty za udzielenie poprawnej odpowiedzi. Zmienną do ich magazynowania – o nazwie **punkty** I i wartości początkowej równej 0 – deklarujemy przed pętlą **for**.

```
punkty = 0 I
for i in a:
    print(i[1])
    print("A) "+i[2])
```

8 Wartość zmiennej **punkty** powinna rosnąć tylko wtedy, kiedy użytkownik poda prawidłową odpowiedź. Powinniśmy zatem dodać instrukcję warunkową **if** na końcu pętli **for**. Warunkiem powinno być to, że w zmiennej o nazwie **udzielona** znajduje się taka sama wartość jak w tabeli, w kolumnie z poprawną odpowiedzią. Jest to ostatnia kolumna, o indeksie 6, zatem zapisujemy całą linię w postaci **if udzielona == i[6]:** J.

```
udzielona = input("Podaj odpowied
if udzielona == i[6]: J
```

9 Instrukcja wykonująca się, gdy warunek jest spełniony, to zwiększenie wartości zmiennej **punkty** o 1 K.

```
udzielona = input("Podaj
if udzielona == i[6]:
    punkty = punkty +1 K
```

10 Po przejściu całej pętli **for** wykonają się instrukcje, jakie zapiszemy w kodzie programu po pętli. Pętla skończy swoje działanie po wyświetleniu wszystkich pytań z bazy. Jeśli chcemy poinformować użytkownika o uzyskanym przez niego wyniku, piszemy po pętli **print("Twój wynik to: "+str(punkty))** L.

```
    if udzielona == i[6]:
        punkty = punkty +1 L
print("Twój wynik to: "+str(punkty))
```

11 Po wyświetleniu wszystkich pytań użytkownik powinien zobaczyć zdobytą liczbę punktów M.

QUIZ W WERSJI Z OKNEM GRAFICZNYM

Grę typu quiz opartą na bazie danych z pytaniami możemy zrealizować nie tylko w formie tekstowej, ale też w oknie graficznym. W tym celu możemy skorzystać z różnych rodzajów okien dostarczanych nam przez rozmaite moduły. Na potrzeby tego poradnika okno zostanie stworzone poprzez często obecny w szkolnej edukacji programistycznej moduł **turtle**. Do rozpoczęcia pracy nad skryptem możemy wykorzystać podstawę programu A napisaną w części Wyciąganie danych z bazy.

```
File  Edit  Format  Run  Options  Window  Help
import sqlite3
conn = sqlite3.connect('e.sqlite')

c = conn.cursor()
a = c.execute("SELECT * from pytania")

for i in a:
    print(i)

conn.close()
```

1 Poprzez **import turtle** B na początku skryptu importujemy potrzebny nam moduł turtle.

```
File  Edit  Format  Run  Options  Window  Help
import sqlite3
import turtle B
conn = sqlite3.connect('e.sqlite')
```

```
turtle.bgpic("tlo.png") C
c = conn.cursor()
a = c.execute("SELECT * from pytania")
```

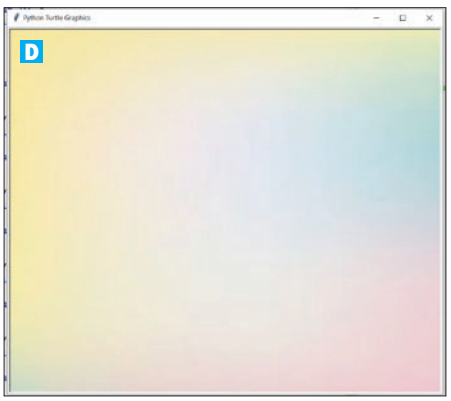
2 Dodajemy polecenie **turtle.bgpic("tlo.png")** C, by okno graficzne w naszym quizie miało tło, gdzie **tlo.png** to nazwa pliku graficznego, jaki chcemy umieścić w tle. Możemy wybrać dowolny plik graficzny według swoich preferencji. Pamiętajmy jednak, by jego wielkość była na tyle duża (najlepiej więcej nić 500 pikseli szerokości i wysokości), by wypełniła okno naszej gry. Zwróćmy także uwagę na to, by plik ten znajdował się w tej samej lokalizacji, w której znajduje się kod gry.

3 Po uruchomieniu programu z wprowadzonymi zmianami zobaczymy okno graficzne wypełnione wybranym tłem D.

4 Dalej możemy utworzyć obiekt żółwia, którego wykorzystamy do wypisania treści pytania i odpowiedzi w oknie. Obiekt tworzymy, pisząc **x = turtle.Turtle()** E.

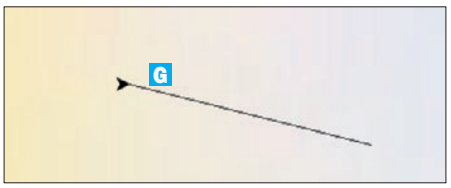
```
turtle.bgpic("tlo.png")
x = turtle.Turtle() E
c = conn.cursor()
```

5 W kolejnej części będziemy edytować zawartość pętli **for**, tak by nie wypisywać pytań w konsoli, ale poprzez utworzonego żółwia o nazwie **x**. Dlatego z pętli kasujemy linię **print(i)** i zastępujemy ją poleceniem **x.setpos(-200,50)** F.




```
for i in a:
  F x.setpos(-200,50)
```

6 Jak widać na ilustracji G, obiekt przenosi się w nowe miejsce, jednak pozostawia po sobie ślad. By śladu nie było widać, dodajemy jeszcze przed pętlą polecenie **x.pu()** H.




```
x.pu() H
for i in a:
    x.setpos(-200,50)
```

Wyrwij, użyj dziurkacza i zbieraj do segregatora

7 Wróćmy do pętli. Po przeniesieniu obiektu **x** w nowe miejsce możemy w pętli użyć polecenia **x.write**, by wypisać treść pytania. Do treści odnosimy się podobnie jak w przypadku quizu pisanego w trybie tekstowym, czyli **x.write(i[1])** I. Po uruchomieniu programu wszystkie pytania wypiszą się na sobie. W dalszej części poradnika będziemy usuwać ten efekt, jednak najpierw zajmiemy się wypisaniem odpowiedzi do wyboru.

```
for i in a:
    x.setpos(-200,50)
    x.write(i[1]) I
```

8 Przed wypisaniem każdej z odpowiedzi przenosimy obiekt **x** w nowe miejsce poprzez **x.setpos** i wypisujemy odpowiedź wraz z odpowiadającą jej literą J. Odpowiedzi,

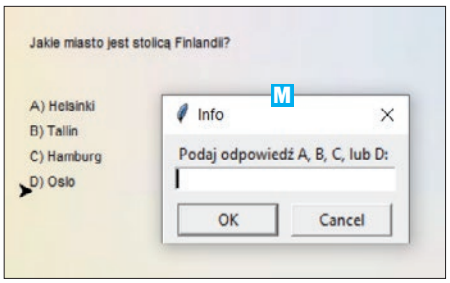
```
for i in a:
    x.setpos(-200,50)
    x.write(i[1])
    x.setpos(-200,0)
    x.write("A) "+i[2])
J   x.setpos(-200,-20)
    x.write("B) "+i[3])
    x.setpos(-200,-40)
    x.write("C) "+i[4])
    x.setpos(-200,-60)
    x.write("D) "+i[5])
```

podobnie jak pytania, na tym etapie realizacji zadania wyświetlają się na sobie K, co uniemożliwia ich prawidłowe odczytanie.

K

```
x.write("D) "+i[5])
x.setpos(-200,-60)
x.write("D) "+i[5])
udzielona = turtle.textinput("Info","Wybierz odpowiedź A, B, C lub D: ") L
```

9 Efektu tego unikniemy, dodając na końcu pętli oczekiwanie na udzielenie przez gracza odpowiedzi, podobnie jak było to w przypadku quizu w trybie tekstowym. Tym razem na końcu pętli **for** dodajemy **udzielona = turtle.textinput("Info","Wybierz odpowiedź A, B, C lub D: ")** L, by program wyświetlał okno pozwalające na podanie odpowiedzi M.



10 Dalej pozostaje nam już tylko dodanie zliczania punktów i sprawdzania poprawności odpowiedzi, co realizujemy analogicznie do gry w trybie tekstowym, tworząc zmienną **punkty** o wartości początkowej równej **0** N przed pętlą **for**, a także dodając

```
x.pu()
punkty = 0 N
for i in a:
    x.setpos(-200,50)
    x.write(i[1])
    x.setpos(-200,0)
    x.write("A) "+i[2])
    x.setpos(-200,-20)
    x.write("B) "+i[3])
    x.setpos(-200,-40)
    x.write("C) "+i[4])
    x.setpos(-200,-60)
    x.write("D) "+i[5])
    udzielona = turtle.text
    if udzielona == i[6]:
      O punkty = punkty + 1
```

instrukcję warunkową **if** na końcu pętli **for**. Warunkiem ponownie powinno być to, że w zmiennej o nazwie **udzielona** znajduje się taka sama wartość jak w tabeli, w kolumnie z poprawną odpowiedzią. Gdy warunek jest spełniony, wartość zmiennej **punkty** powinna wzrosnąć o **1** O.

11 Wartość zmiennej **punkty** powinna być wypisana w oknie poprzez obiekt **x** po przeniesieniu go w nowe miejsce. Dodajemy zatem polecenia **setpos** i **write** P.

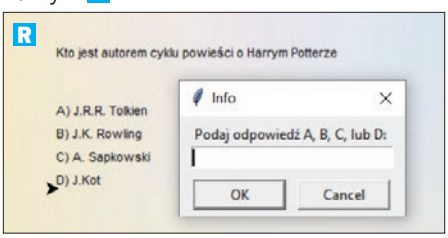
```
    if udzielona == i[6]:
P       punkty = punkty + 1
x.setpos(-200,-80)
x.write("Twój wynik to: "+str(punkty))
```

12 Gra cały czas wyświetla jednak treść pytań i odpowiedzi na sobie. Przed wypisaniem kolejnego pytania powinniśmy jeszcze czyścić zawartość okna z treści poprzedniego pytania. Efekt ten uzyskamy, pisząc **x.clear()** Q na końcu pętli **for** po sprawdzeniu poprawności udzielonej odpowiedzi.

```
    udzielona = turtle.text
    if udzielona == i[6]:
        punkty = punkty + 1
Q   x.clear()
x.setpos(-200,-80)
```

13 W ten sposób mamy już w pełni działający quiz zrealizowany w oknie graficznym R.

R



WYKORZYSTANIE BAZY DANYCH DO PRZECHOWYWANIA LISTY NAJLEPSZYCH WYNIKÓW

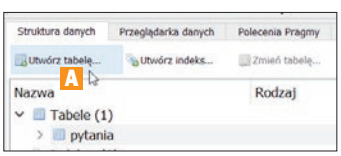
Wiele gier, z jakimi spotykamy się na co dzień, pozwala na przechowywanie listy najlepszych wyników. Lista ta jest aktualizowana wraz z kolejnymi rozgrywkami, a wyniki przechowywane są nie tylko od ostatniego uruchomienia gry, ale zdecydowanie dłużej. Gdybyśmy taką listę wyników chcieli zrealizować, opierając się na zwykłych zmiennych czy też listach, ich zawartość resetowałaby się przy kolejnych uruchomieniach gry. Jeśli oprzemy listę wyników na tabeli z bazy danych, będzie ona przechowywana w zewnętrznym pliku, zatem wyniki nie będą z niej znikać przy kolejnych uruchomieniach gry. Do realizacji listy wyników wykorzystamy program z quizem w wersji graficznej.

TWORZENIE TABELI

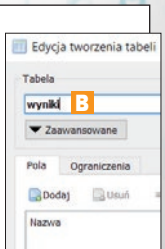
Wiemy już, że tabele w bazie danych można stworzyć z poziomu kodu źródłowego, ale można to też zrobić z poziomu programu **DB Browser for SQLite** – poznajmy ten sposób.

1 Otwieramy w programie bazę danych, w której przechowywane są pytania.

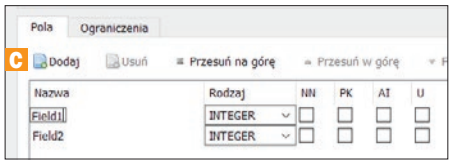
2 Klikamy na przycisk **Utwórz tabelę** A.



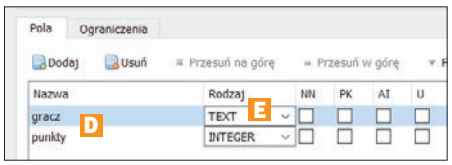
3 W wyświetlonym oknie tworzenia tabeli w pole **Nazwa** wpisujemy **wyniki** B.



4 Dwukrotnie klikamy na przycisk **Dodajemy** C, by w tabeli utworzyć dwie kolumny.



5 Uzupełniamy tabelkę poniżej przycisku tak, by w kolumnie **Nazwa** pojawiły się



wpisy **gracz** i **punkty** D, a w kolumnie **Rodzaj** przy pozycji **gracz** zmieniamy **Integer** na **Text** E.

F

```
CREATE TABLE "wyniki" (
    "gracz" INTEGER,
    "punkty"    INTEGER
);
```

G OK Anuluj

6 Podczas uzupełniania tabeli generowane jest polecenie SQL **Create Table** F, które umieszczone w kodzie programu utworzyło-

by taką samą tabelę. Gdy okno tworzenia tabeli jest już uzupełnione zgodnie z opisem, klikamy na **OK** G.

7 Po utworzeniu tabeli kombinacją klawiszy ctrl+S zapisujemy zmiany wprowadzone do bazy.

ZAPISYWANIE WYNIKÓW W BAZIE

1 Dalej przechodzimy do kodu gry. W nim po wyświetleniu wyniku uzyskanego przez gracza możemy przejść do zadania graczowi pytania o to, pod jaką nazwą chciałby zapisać w bazie swój wynik. Możemy to zrobić linią kodu **nazwa = turtle.textinput("Zapisz wynik","Podaj nazwę gracza:")** A.

```
    if udzielona == i[6]:
        punkty = punkty + 1
    x.clear()
x.setpos(-200,-80)
x.write("Twój wynik to: "+str(punkty))
nazwa = turtle.textinput("Zapisz wynik","Podaj nazwę gracza:")
```

A

2 Następnie dodajemy w kodzie programu linię **c.execute("INSERT INTO wyniki VALUES (' "+nazwa+" ',"+str(punkty)+")")** B, która wywołuje polecenie wstawiające dane do tabeli **wyniki**. Zostanie tam wstawiona podana wcześniej nazwa gracza oraz uzyskana w bieżącej rozgrywce liczba punktów.

```
x.write("Twój wynik to: "+str(punkty))
nazwa = turtle.textinput("Zapisz wynik","Podaj nazwę gracza:")
c.execute("INSERT INTO wyniki VALUES ('"+nazwa+"',"+str(punkty)+")")
```

B

3 Dodajemy jeszcze polecenie **conn.commit()** C przed zamknięciem połączenia z bazą da-

```
c.execute("INSERT
conn.commit() C
```

nych, by zatwierdzić zmiany w bazie w postaci wprowadzenia danych do tabeli.

Wyświetlanie listy wyników

Wyświetlanie listy wyników można zrealizować, dodając do gry kod przedstawiony na ilustracji obok, wstawiając nowe linie między **conn.commit()** a **conn.close()**. W przedstawionym kodzie poszczególne linie oznaczają:

A **x.clear()** – wyczyszczenie wcześniej widocznej treści okna graficznego.

B **b = c.execute("SELECT * from wyniki Order by punkty desc")** – zapisanie pod nazwą **b** wszystkich danych z tabeli z wynikami. Dodanie fragmentu **Order by punkty desc** w poleceniu **Select** sprawia, że wyniki będą posortowane malejąco względem uzyskanej liczby punktów.

C **x.setpos(0,250)** – ustawienie obiektu **x** (żółwia) w górnej części okna gry.

D **for i in b:** – pętla przechodząca po wszystkich wierszach danych wyciągniętych z tabeli.

E **x.write(i[0])** – wypisanie nazwy gracza.

F **x.fd(100)** – przesunięcie żółwia do przodu przemieszcza go w prawą stronę w oknie gry.

```
nazwa = turtle.textinput("Zapisz wynik","Podaj nazwę gracza
c.execute("INSERT INTO wyniki VALUES ('"+nazwa+"',"+str(pun
conn.commit()
A x.clear()
B b = c.execute("SELECT * from wyniki Order by punkty desc")
C x.setpos(0,250)
D for i in b:
    E x.write(i[0])
    F x.fd(100)
    G x.write(i[1])
    H x.bk(100)
      x.right(90)
      x.fd(20)
      x.left(90)

conn.close()
```

G **x.write(i[1])** – wypisanie wyniku uzyskanego przez gracza.

H **x.bk(100)**, **x.right(90)**, **x.fd(20)**, **x.left(90)** – kolejne linie umieszczone w pętli sprawiają, że obiekt **x**, czyli żółw, przestawia się nieco niżej, tak by w kolejnych przejściach pętli nazwy gracza i liczba uzyskanych przez niego punktów wypisana była niżej niż wcześniej.

Gotowa lista wyników może wyglądać jak na ilustracji poniżej.

Wyrwij, użyj dziurkacza i zbieraj do segregatora